AFRL-RI-RS-TR-2012-196

MONTAGE: A METHODOLOGY FOR DESIGNING COMPOSABLE END-TO-END SECURE DISTRIBUTED SYSTEMS

INTERNATIONAL BUSINESS MACHINES CORPORATION

AUGUST 2012

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2012-196 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

PATRICK HURLEY
Work Unit Manager

/ S /

WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
AUG 2012	FINAL TECHNICAL REPORT	MAR 2008 – MAR 2012

4. TITLE AND SUBTITLE
MONTAGE: A Methodology for Designing Composable End-to-End Secure Distributed Systems

5a. CONTRACT NUMBER
FA8750-08-2-0091

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Suresh Chari

5d. PROJECT NUMBER
DHS3

5e. TASK NUMBER
IB

5f. WORK UNIT NUMBER
M1

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
International Business Machines Corporation
Thomas J. Watson Research Center
1101 Kitchawan Road
Yorktown Heights NY 10598

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-196

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This report describes the Montage project, a principled approach to build secure distributed systems which remain secure when composed with other systems. This is an application of the Universal Composability Framework, which has been previously applied successfully to design cryptographic protocols, to the problem of designing software systems. This report describes how the framework can to be adapted to apply to software systems. Further it describes the successful application of this new framework to diverse applications including the design of safe subsets of the POSIX file system interface, the design of secure virtualization primitives and the analysis of web protocols. We also describe an attempt to automate the use of this framework by automatically generating proofs of equivalence required in application of this framework. Our results show that it is feasible to design large composably secure software systems using this framework.

15. SUBJECT TERMS
Secure systems, provable security, composability, reusable secure components

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					PATRICK HURLEY
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPONE NUMBER (*Include area code)*
U	U	U	SAR	202	**N/A**

Abstract

This report describes the Montage project whose goal is to develop methodologies for the design of the secure composable systems. This project aims to extend the Universally Composable Security Framework [Can], a security–via–composition methodology used extensively for designing secure cryptographic protocols, and apply it to the design of more realistic and practical systems. This report describes the successful application of the framework to the design of a number of different software systems as well as the results of our attempts to automate the application of this framework.

First, we adapt the UC Framework by establishing new conventions for modeling process management and scheduling and use this to guarantee basic integrity properties of a POSIX-like filesystem.. Our main contribution is a very simple filesystem specification model, called SimpFS, that captures many integrity concerns in contemporary filesystems, together with an implementation over existing POSIX filesystems and a proof that the implementation realizes the specification. The composability properties of our analysis imply that any software system that uses our implementation over POSIX behaves essentially the same as if it were using the simple, idealized specification system. This equivalence required the development of a new filesystem primitive such as safe-open which is a drop-in replacement of the POSIX open. Experiments on several UNIX variants suggest that this solution can be deployed in a portable way without breaking existing systems, and that it is effective against a whole class of pathname resolution attacks. This evaluation revealed a number of latent vulnerabilities in components such as the CUPS daemon, Fedora Core init scripts, Tomcat servers as well as MySQL and XAMPP servers, *all* of which can be corrected using safe-open.

Another application of UC is to secure virtualization primitives: We propose an intuitive model of isolation, equivalent to executing on physically separate hardware, and derive practical requirements for hypervisors on shared hardware to achieve such isolation. We use UC to prove that these requirements are sufficient for isolation. Further, we explore how commodity virtualization platforms can realize the isolation requirements we have identified. We believe that basing the notion of isolation on a provably secure models such as UC will be a significant step towards comprehensive formal models for isolation.

A key thrust of the Montage project has been to make the task of writing the formal proofs of equivalence needed more practical. To this end we undertook a very ambitious effort to achieve automatable proofs of security in the UC framework. First we note that the general problem is undecidable as the problem of automating UC proofs is identical to deciding if two programs are the same. We looked at restrictions on the language used to describe the real and ideal systems as a way of ensuring automation of proofs of equivalence. We started with really simple straight line programs with which we can demonstrate automatability and then gradually expanded the expressive power of the language classes: In particular we consider language classes which are inspired by formulations of standard cryptographic primitives. In considering these language classes, we have obtained both negative and positive results.

We believe that the results of this project clearly demonstrate that it is indeed feasible to base the design of composably secure large practical systems on formal frameworks such as Universal Composability.

Contents

List of Figures

List of Tables

1 Summary

The overall goal of the Montage project was the development of a methodology for the design of the secure composable systems. This project aims to extend the Universally Composable Security Framework (UC) [Can], a security–via–composition methodology used extensively for designing secure cryptographic protocols, and apply it to the design of more practical and realistic software systems.

The key objectives of the project were to find appropriate large software systems to model using this framework. In particular, two such systems which were identified as part of the original proposal were the Postfix mail server and the concept of a Trusted Virtual Domains (TVD). Postfix is a very popular mailer daemon which is a drop in replacement for sendmail. It is designed with security in mind and has a number of carefully designed features to avoid known pitfalls. We have successfully applied the UC framework to model some of these features of Postfix including the mechanisms by which it handles files in world-writable directories. This has led to a very general primitive as described in the accomplishments. We have also extended the UC Framework to model secure virtualization primitives. This is a key building block to composably building secure and trusted virtual domains which co-exist on the same physical platforms.

Another key objective of the Montage project was to make the application of the methodology practical by outlining methodologies where the complex proofs of equivalence required to apply this methodology could be automated. We undertook a fundamental study of which proofs in the UC Framework could be automated and obtained foundational results on the subsets of the specification language which yield automatable proofs. An investigation into these methods has also yielded more efficient cryptographic primitives. These results are detailed in the accomplishments.

Overall, we have achieved most if not all of the ambitious objectives that we set out for the Montage project. We showed many convincing examples in the design of secure software and systems that the UC methodology can be successfully applied to the design of large software systems. We have shown that a principled approach yield practical primitives which are designed to be composable in arbitrary environments. Our safe file system open primitive can be a drop in replacement for the usual POSIX open and using that prevents hundreds of vulnerabilities reported in the last few years. We have used this methodology to identify potential vulnerabilities in many open source products and transfered our technology to these software modules. The work on methodologies to automate proofs is foundational and will lead to theorem-prover based automation software for automation of proofs and lead to broader applications of the UC Framework.

1.1 Key Accomplishments

The following are some of the key accomplishments of the Montage project. The details of these and other contributions of the project will be described in the rest of this technical report.

- **Composable Security Analysis of Operating System Services** While the UC Framework has been applied extensively to the design and analysis of cryptographicprotocols, a key advance we have made is to adapt the framework so it can be applied to the design of software systems. In particular, we looked at abstracting some of the security features of the Postfix mail server. As a test case, we modeled how Postfix safely uses the POSIX filesystem interface, designed an ideal safe filesystem interface and showed its equivalence to a realization over POSIX. This demonstrates the successful application of the UC framework to a large practical system *i.e.* the POSIX filesystem interface. The UC framework, which is explicitly designed for composability, gives us the following strong security guarantee: If all the uncorrupted processes in the system use our filesystem interface then, *irrespective* of what corrupted process do using the *entire* POSIX interface, good processes are *guaranteed* to be secure against a class of privilege escalation attacks. The compositionality theorem in the UC Framework guarantees that this property is retained *irrespective of what larger program the processes are implementing*. To the best of our knowledge this is one of the few cases of successfully modeling a large interface such as POSIX. We conclude from this experience that the UC Framework can be applied to model practical software systems. A technical paper resulting from this effort was published at the Applied Cryptography and Network Security conference in June 2011.

- **A secure filesystem primitive** One of the results of the formal modeling of the safe use of POSIX was the development of a practical filesystem primitive which can be used securely in any software system. We developed primitives for a POSIX environment, providing assurance that files in safe directories (such as /etc/passwd) cannot be opened by looking up a file by an unsafe pathname (such as a pathname that resolves through a symbolic link in a world-writable directory). In todays UNIX systems, solutions to this problem are typically built into (some) applications and use application specific knowledge about (un)safety of certain directories. In contrast, our solution can be implemented in the filesystem itself (or a library on top of it), thus providing protection to all applications. Our solution is built around the concept of pathname manipulators, which are roughly the users that can influence the result of a file lookup operation. For each user, we distinguish unsafe pathnames from safe pathnames according to whether or not the pathname has any manipulators other than that user or root. We propose a safe-open procedure that keeps track of the safety of the current pathname as it resolves it, and that takes extra precautions while opening files with unsafe pathnames. We prove that our solution can prevent a common class of filename-based privilege escalation attacks, and describe our implementation of the safe-open procedure as a library function over the POSIX filesystem interface. We tested our implementation on several UNIX variants to evaluate its implications for systems and applications. Our experiments suggest that this solution can be deployed in a portable way without breaking existing systems, and that it is effective against this class of pathname resolution attacks. This work resulted in a publication which appeared in the Network and Distributed systems security conference in Feb 2010.

- **Technology transfer to Open Source Software** As mentioned above, we have validated the applicability of our filesystem primitive against *all* processes in common Linux variants. In the process of evaluation we allowed all accesses but flagged what we deemed as unsafe accesses which our safe-open primitive would have disallowed. From this evaluation we had identified a number of latent vulnerabilities in a number of important software packages such as the CUPS(Common Unix Print Services) daemon, Fedora Core init scripts, important servers such as Tomcat, open source packages such as MySQL and the XAMPP daemon as well as other software packages. In each case we filed the appropriate vulnerability report as well as highlighted how our approach using *safe-open* could uniformly solve this problem. This has been successfully adopted by a number of open source software including the CUPS daemon and Fedora Core init scripts.

- **Secure Virtualization Primitives** Another key area where we have applied the UC methodology is the domain of secure virtualization. As virtualization becomes increasingly popular in both cloud and personal computing, there is an acute need for strong isolation between virtual machines. Yet such isolation is not achieved in current implementations and lacks a formal model sufficiently expressive for contemporary commodity systems. On one hand existing virtualization platforms have been shown to have significant side channels via shared resources (CPUs, caches, disks, etc.). On the other hand well-established security models of isolation are developed around abstract models of computation and do not easily translate to the complex hardware architectures of today. We propose an intuitive model of isolation, equivalent to executing on physically separate hardware, and derive five practical requirements for hypervisors on shared hardware to achieve such isolation. We use the Universal Composability (UC) framework to prove that these requirements are sufficient for isolation. Further, we explore how commodity virtualization platforms can realize the isolation requirements we have identified. We relate timing side channels to scheduling policies and show that only a small, well-defined subclass of schedulers eliminate timing-based leakage. An implementor that uses such schedulers is guaranteed to have a hypervisor free of timing side channels. We also show how one can trade scheduling optimality for limited isolation by quantifying the rate of leakage in probabilistic schedulers. We analyze the root cause of explicit information flows and demonstrate that access control (via address translation or access matrices) and correct operation of execution context save and restore operations is sufficient to prevent explicit information flows. Since our formal models focus on a proof-driven methodology to realize isolation in commodity virtualization platforms, we believe that ours is a significant step towards comprehensive formal models for isolation.

- **Proofs of security of Web Security Protocols** Another broad area where we have tried to apply the UC Framework is that of web security protocols. These are good examples of protocols which are mainly cryptographic in nature but include enough systems and web artifacts to make it a complex application of the UC Framework.

We focused on modeling OAuth, the popular web security protocol for delegation. We first looked at OAuth1.0 and wrote a formal model using the UC framework. A bug which was discovered in the protocol, falls out naturally from our formal model. Since then, IETF is standardizing the next version of the protocol, OAuth 2.0. To provide guarantees of correctness, we have written formal models of this protocol. From our analysis, we have derived recommendations for implementation which are necessary for security.

- **Proof Automation Methodologies** Another key effort in the Montage project has been to make the task of writing the formal proofs of equivalence needed more practical. To this end we undertook a very ambitious effort to achieve automatable proofs of security in the UC framework. First we note that the general problem is undecidable. This is because at some level the problem of automating UC proofs is identical to deciding if two programs are the same and that is clearly undecidable. Thus our approach is to look at restrictions where we can automate the proofs of equivalence. These restrictions on the real and ideal functionalities are define by restricting the languages which we can write the real and ideal systems. Thus the restricted problem is to automate the equivalence of real and ideal systems where both the systems are described in a restricted language class. We start with really simple straight line programs with which we can demonstrate automatability. Then we gradually expand the expressive power of the language classes: In particular we consider language classes which are inspired by formulations of standard cryptographic primitives. In considering these language classes, we have obtained both negative and positive results.

  1. **Negative results**: If both the ideal and real systems are allowed to be arbitrary Turing Machines, then, as noted earlier, the problem becomes as hard as deciding program equivalence, which is undecidable. Further, we showed that even under somewhat more restricted scenarios the problem remains undecidable. For instance if we consider programs which are allowed unbounded table lookup/storage operations (not random access, but just storing and detecting whether some string is there in the table) then the problem is undecidable even when the program is otherwise restricted such as when there are no arithmetic operations, loops in subroutines or random number generation.
  2. **Positive results**: Despite these negative results we show that there are other restricted languages which we can obtain automation procedures. The restrictions for which we have positive results avoiding the undecidable territories and for which we can demonstrate automation are programs written in a language that allows environment input/outputs, adversary send/receives, conditional execution, random number generation, uninterpreted functions and bounded local storage. As we demonstrate this is sufficient to model several important cryptographic primitives.

  Our work is the first to investigate the automatability for proofs in the UC Framework. Our work has charted a portion of the boundary between what is provably undecidable

and provably decidable. The eventual goal is to cover automated analysis of a wide range of interesting cryptographic protocols. The positive results from our work covers language classes that are sufficient to cover the description of strong cryptographic primitives such as public key encryption, digital signatures and password-based key exchange protocols. However, to cover more complex cryptographic protocols like OAuth and systems protocols like file systems with safe-open, we need to provide support for arbitrary serialization of the real protocol execution. A paper describing results of our proof automation work will be presented at the European Symposium on Research in Computer Security (ESORICS) in Sep. 2012.

- **Efficient Cryptographic primitives** Arising from this investigation of the automation of UC proofs, we have investigated efficient password based key exchange protocols. These protocols are expressible in the limited language for which we have been successful in automating the proofs. We have discovered novel schemes for this primitive both in the UC model of security as well as other models such as PAK. We have introduced the notion of relatively-sound non-interactive zero knowledge(NIZK) protocols which we use as a building block for efficient password based key exchange primitives. As a corollary we obtain efficient implementations of other cryptographic primitives and show several efficient implementation of this primitive in the UC models. This investigation of password based key exchange protocols is motivated with finding non-trivial protocols where we can automate the proofs of the application of the UC framework. This work will be published in the Public Key Cryptography conference in May 2012.

2 Introduction

The design of systems with a desired set of security properties and the reasoning that the system satisfies the security properties is a notoriously difficult task. Formally specifying the security properties of a system and proving that the system satisfies these formal specifications is typically unmanageable, for all but the smallest of applications. Contemporary software systems are complex, consisting of many millions of lines of code, spread across a myriad of components and sub-components. It is therefore tempting to break a system into components and reason about the security properties of the smaller components. This approach too is problematic, however, since in general a composition of secure components does not directly result in a secure system: there is usually no guarantee that the security properties of individual components are preserved under composition. Often, a component will be used in environments different from what its designers initially had in mind, alongside other components that perhaps did not even exist when the original component was analyzed, potentially violating some assumptions that were made in the analysis.

Ideally, we would like to analyze the behavior of a component in isolation, and have the assurance that this behavior remains intact even when that component is embedded in a new environment. This approach is possible in the mathematically precise realm of cryptography and cryptographic protocols through several frameworks such as Universal Composability (UC)[Can01, Can06] and closely related approaches such as Reactive Simulatability[PW00, BPW07]. These frameworks are aimed at capturing the security of cryptographic primitives and protocols, ranging from authentication and key exchange, to public-key encryption and signatures, zero-knowledge, and more (see [Can06] for many examples.) However, many of the features of these frameworks appear at first to be specific to the realm of cryptographic protocols. A natural question is whether the "composable security" approach can be carried out in a meaningful way outside the limited domain of cryptography. In particular:

> *Can we obtain meaningful composable security in the context of general software systems?*

The goal of the Montage project is to establish a new methodology for designing secure systems through (de)composition. Our goal is to identify a set of processes, patterns and potentially tools to enable the design of composable secure systems. The foundation of our methodology is the Universally Composable Security framework, which we aim to extend by introduce new elements in all the software engineering aspects of secure system design: processes, patterns and tools.

The Universally Composable Security framework allows for specifying security properties of cryptographic protocols in a clear and concise way and provides tools for asserting whether a given protocol satisfies the specification. The key result of this framework is a very effective way for reducing the complexity of security analysis, via a strong security preserving composition theorem. It thus provides a way for decomposing large systems into small components, separately analyzing the security properties of each component, and then

deducing the security properties of the composite system from the security properties of the components.

The Montage project aims to take the UC methodology, which has proven very useful in designing secure multiparty cryptographic protocols, and scale it out to apply to more realistic and complex systems. The main advantage provided by the UC framework is that a component designed securely in it will retain its desired security features independent of the higher level system that it is used in. This approach has very quickly become an important design pattern for secure multiparty cryptographic protocols. We wish to scale this approach up and extend it to deal with complex (and more complete) systems. The desired outcome of the Montage project would be to make the UC framework an essential tool in the design of secure software systems.

First we will describe the UC Framework which is the formal underpinning to the entire project Then we describe the targeted use cases and the rationale for choosing them.

2.1 The Universal Composability Framework

We briefly describe the relevant aspects of the framework of universally composable (UC) security. The reader is referred to [Can01] for more details. The framework describes two probabilistic games: The *real world* that captures the protocol flows and the capabilities of an attacker, and the *ideal world* that captures what we think of as a secure system. The notion of security asserts that these two worlds are essentially equivalent.

The real-world model. The players in the real-world model are all the entities of interest in the system (e.g., the nodes in a network, the processes in a software system, etc.), as well as *the adversary* A and *the environment* $\mathcal{Z}$. All these players are modeled as efficient, probabilistic, message-driven programs (formally, they are all interactive Turing machines).

The actions in this game should capture all the interfaces that the various participants can utilize in an actual deployment of this component in the real world. In particular, the capabilities of A should capture all the interfaces that a real-life attacker can utilize in an attack on the system. (For example, A can typically see and modify network traffic.) The environment $\mathcal{Z}$ is responsible for providing all the inputs to the players and getting all the outputs back from them. Also, $\mathcal{Z}$ is in general allowed to communicate with the adversary A. (This captures potential interactions where higher-level protocols may leak things to the adversary, etc.)

The ideal-world model. Security in the UC framework is specified via an "ideal functionality" (usually denoted $\mathcal{F}$), which is thought of as a piece of code to be run by a completely trusted entity in the ideal world. The specification of $\mathcal{F}$ codifies the security properties of the component at hand. Formally, the ideal-world model has the same environment as the real-world model, but we pretend that there is a completely trusted party (called "the functionality"), which is performing all the tasks that are required of the protocol. In the ideal world, participants just give their inputs to the functionality $\mathcal{F}$, which produces the correct outputs (based on the specification) and hands them back to the participants. $\mathcal{F}$ may interact with an adversary, but only to the extent that the intended security allows.

(E.g., it can "leak" to the adversary things that should be publicly available, such as public keys.) Specifying the code of $\mathcal{F}$ is typically a non-trivial task. It is important that $\mathcal{F}$ satisfies all the desired security properties, but also that $\mathcal{F}$ does not impose unnecessary constraints: It is only too easy to write a functionality that describes "what we intuitively want", but is not realizable by any implementation.Another crucial concern is the *simplicity* of the functionality $\mathcal{F}$, since we want $\mathcal{F}$ to capture the important security concerns, not the mundane implementation details.

UC-Security and the Composition Theorem. An implementation π **securely realizes** an ideal functionality $\mathcal{F}$ if no external environment can distinguish between running the protocol π in the real world and interacting with the trusted entity running the ideal functionality $\mathcal{F}$ in the ideal world. That is, for every adversary A in the real world, there should exist an adversary A' in the ideal world, such that no environment $\mathcal{Z}$ can distinguish between interacting with A and π in the real world and interacting with A' and $\mathcal{F}$ in the ideal world.

The striking feature of the UC framework is its ability to handle composition. Specifically, the composition theorem from [Can01] asserts the following: Let ρ be an arbitrary system that runs in the ideal world and uses (perhaps multiple copies of) the functionality $\mathcal{F}$. Next, consider the system ρ' in the real world, that is the same as ρ except that in ρ' each call to the ideal functionality $\mathcal{F}$ is replaced by executing the implementation π. Then, if π securely realizes $\mathcal{F}$ it is guaranteed that system ρ' behaves essentially the same as system ρ. In particular, all the security properties of ρ are inherited by protocol ρ'. This guarantee is the basis for the composable security guarantees provided by the UC framework.

2.2 Systems targeted by this methodology

While we expect that frameworks such as Universal Composability (UC) should be applicable to the design of a wide variety of software systems, we have focused our efforts on the following specific types of systems which we expect to have the maximum impact.

Modularly designed software systems: The best candidates for the UC Framework are, of course, software systems which are inherently designed as small components with cleanly defined interfaces between the components. A very good example of such a system is the Postfix[Ven] mail server. It consists of clearly delineated components which interact in a well defined way. This is an important test case for the applicability of the UC Framework for several reasons: it is a complex large system yet it is designed in a modular componentized manner, it is a noncryptographic system and would therefore be useful in refining the UC Framework and lastly, its size would give us a way to calibrate the scalability of the framework. Our first results were obtained by taking some of the security features of the Postfix system such as its careful handling of the filesystem and abstracting it. The extensions of the UC Framework to support this and the resulting formal models and implementations are presented in Sections 5 and 6.

Trusted Virtual Domains: Trusted Virtual Domains (TVDs) represent a new model for IT security by providing explicit infrastructurelevel containment and trust guarantees.

TVD enables the abstraction of shared physical infrastructure into isolated and contained virtual domains. TVDs can be thought of as a trusted and isolated distributed infrastructure which can be built modularly by adding components such as compute, storage, partitioned applications etc. This view supports the modeling of these systems using a composable security framework since we don't want to redo the security analysis every time a new virtual component is added. In the Montage project, we have studied the application of UC to the design of secure virtualization primitives. Section 7 describes the application to obtaining practical requirements on hypervisors which guarantees strong notions of isolation.

Web Security Protocols: Web Security protocols are great candidates for modeling with composition frameworks such as UC. First these protocols involve elements such as browser implementations, infrastructures components such as DNS in addition to core cryptographic primitives. Thus modeling them would require only a slight extensions of the core framework but would extend it in a manner requiring analysis of software implementations. Secondly, web security protocols are arbitrarily composed with other protocols. For instance SSL is used with almost every other protocol to obtain secure variants. Section 8 describes practical examples of some composition failures that were detected out of the work done in the Montage project. Section 9 describes our effort to formally model the OAuth2.0 web security protocol.

2.3 Automating Proofs of Equivalence

Typically, secure software design methodologies such as UC require the user to write down formal proofs. Security in the UC framework requires the software developer to write a formal proof of equivalence between the real and ideal worlds. This is typically a fairly complex task as it requires the analysis of all possible serializations of events in the system of interest. To alleviate this, one of the key goals of the Montage project was to investigate if it is feasible to automatically generate the proof of equivalence. If this is feasible, then developers are much more likely to consider using the rest of the formal machinery. First, we note that the general problem is undecidable in the strong sense as it requires us to automatically decide if two programs are equivalent. In Section 10 we describe several positive results arising from our investigation of special cases where the proofs are amenable to automation. Ours is the first work to consider the automation of proofs in the UC Framework and the positive results we obtain are the first step towards more general automation approaches.

3 Methods, Assumptions and Procedures

This section presents an overview of the methods, assumptions and procedures used in the Montage project. Since the project aims to demonstrate the feasibility of applying the Universal Composability Framework to model software components in a number of diverse areas, we will leave the finer details of the methods, procedures and assumptions for each application to later sections. Here we present a broad overview which captures the essence of the methodology.

3.1 Overview of the UC methodology

The overall goal of the Montage project is the development of a principled methodology for the design of the secure composable systems. In particular, the project aims to extend the Universally Composable Security Framework [Can], a security–via–composition methodology used extensively for designing secure cryptographic protocols, to apply it to more complete and realistic systems.

We have applied this framework to the following diverse application areas to demonstrate the feasibility of the approach:

- The first application we describe in Sections 5 and 6 is to model the safe use the POSIX file system interface. The goal of this work is to design a safe subset of the POSIX file using which we can avoid a large class of attacks due to pathname manipulation.
- Another application of the framework we have developed is that of secure virtualization primitives. This application is described in Section 7. Here we aim to develop a formal model of secure isolation of virtualized environments.
- We have also extended the UC Framework to study the security of common protocols used in web security. Section 8 describes a general class of protocol composition errors that were discovered in a number of common protocols used in the Internet. Section 9 describes the use of the UC Framework to model the security of the Internet protocol OAuth which is increasingly becoming popular in modern web applications.

For each of the target system we have identified, broadly speaking, the steps in applying the Universally Composable security framework to the design and analysis of the target system are:

- Decompose the specified target system into components,
- Identify the abstract functionality as prescribed by the UC Framework as well as the high-level design of the implementation,
- Prove the closeness of the high-level design to the abstract functionality and implementation of the abstract design.

In all the applications of the UC Framework to realistic systems, we have typically needed an iterative process where we revisit the componentization, modeling and proof steps multiple times to ensure the successful application to modeling the system.

By design, we have attempted to apply the framework to very diverse systems from very different systems. Thus the details of each of these use cases *i.e.* the specific components we model, the abstract functionality or the "security specification" for these components and the proof that our implementation realizes the specification will be described when we consider each of the above use-cases.

3.2 Assumptions

The UC methodology for designing software systems essentially systematizes the principle of writing in the specification all the security assumptions we make in the system under test *i.e.* anything that is not explicitly stated in the specification will be protected against in the implementation. For example, for encrypted communication the specification might simply say that we reveal the length of the message to the adversary. The actual implementation of encrypted communication must ensure that nothing other than the length is revealed to the adversary.

Thus writing down precisely the assumptions we make in each target system we investigate is a very crucial task in applying the UC Framework. There is a delicate balance between making the specification too coarse, and thus impossible to realize in a practical system, and making the specification too fine grained which results in all the assumptions being carried over to the larger system this component is embedded in. In each of the use cases we have studied there is an iterative process to precisely capture the assumptions to be written into the specification.

Typically the security assumptions in the specification are written by having an adversary in the ideal world which is then granted powers to intervene and change the order of execution. For instance, if the specification wanted to capture denial-of-service *i.e.* this is not explicitly prevented in the implementation then the functional calls in the specification would include a call to the adversary and execution returns only if the adversary returns control.

The specific assumptions for each of the use cases will be described when we consider each application. Sections 5 and 6 will describe assumptions on various conditions such as denial of service and the ability of the adversary to corrupt specific process which guide our specification. Section 7 describes assumptions such as elementary properties of computer architectures which are needed to prove our conditions for isolation of virtualized components. Section 9 describes assumptions we make on implementations of common Internet mechanisms such as DNS which are needed to argue about security of the OAuth2 protocol.

3.3 Procedures

As described in Section 3.1 a key step in the application of the UC Framework is a proof that the implementation in the real world provably realizes the specification in the idealized world. This equates to proving that, as seen externally by the environment, any behavior that is realizable in the real world can be realized in the ideal world and vice-versa. As with all formal frameworks for secure software design this is often the most difficult task and is typically what leads to the poor applicability of formal frameworks.

It has been a crucial goal of the Montage project to tackle this problem head on with a very ambitious program of automating such proofs of equivalence. Section 10 describes our work on automating proofs of equivalence in the application of the UC Framework. While the general problem can be shown to be undecidable there are significant subsets for which we can describe effective procedures to prove equivalence.

4 Results and Discussion

We have successfully applied the UC Framework to a number of diverse areas covering a wide variety of practical software systems at all levels including the POSIX file system interface, hypervisor implementations, as well as application level protocols such as the web security protocol OAuth2.0. The application of the UC Framework and the specific results obtained in each of these areas in described in subsequent sections. In particular

- Section 5 describes our adaptation of the UC Framework to enable modeling of software systems. The key results in this section include artifacts to model process management and other kernel functions within the UC Framework. We use these adaptations to describe a safe subset of POSIX, using which users are guaranteed to be protected against a class of filename manipulation attacks. The composability guarantees of the UC Framework ensure that these security guarantees are maintained irrespective of what higher level application the process is embedded in.

- Section 6 describes the evaluation of an implementation of the safe filesystem primitives described in Section 5. We show that our primitives can be used as drop in replacements for standard POSIX primitives and are transparently able to run entire Linux systems with it. In the process of evaluating these systems, we have identified latent privilege escalation vulnerabilities in a number of open source software packages.

- Section 7 describes our application of the UC Framework to model isolation in commodity hypervisor environments. We obtain sufficient conditions for such hypervisors running on shared hardware to achieve such isolation. Our models for isolation take into account issues such as timing side channels and other more complex information flows.

- Section 8 describes our investigation of the STARTTLS composition bug which affects a number of open-source software packages implementing common Internet protocols such as SMTP, FTP etc. over TLS.

- Section 9 describes the application of the UC Framework to model web security protocols. We formally model the OAuth2.0 protocol for secure delegation and derive a number of recommendations for implementers which will be necessary for a secure realization of the protocol.

- Section 10 describes our investigation into automation of the proofs of security required for the application of UC Framework. We have obtained a number of positive and negative results on proof automation. The general problem is not decidable but for a number of important subsets we show that it is feasible to automate the proofs of security.

As described in Section 11 this work has resulted in the publication of several technical papers as well as contributions to multiple open source packages.

5 Composable Analysis of Operating System Services

The first target for modeling and design of composably secure components is the Postfix system. As discussed in Section 2, this is an example of a modularly designed software system built with security in mind. We want to abstract out some of the key security features of Postfix. In particular, we look at the safeguards built into Postfix to safely handle files in the system especially in world writable directories. In this section, our main contribution is a very simple filesystem specification model, called SimpFS, that captures many integrity concerns in contemporary filesystems, together with an implementation over existing POSIX filesystems [IEE08] and a proof that the implementation realizes the specification model.

> *The composability properties of our analysis imply that software systems that use our implementation over POSIX behave essentially the same as if they were using the simple, idealized specification system* SimpFS.

This is a very strong security guarantee. In particular, it allows analyzing software systems without worrying about how the filesystem is implemented, and without worrying about potential bad interactions between the analyzed system and the filesystem implementation.

Our filesystem model is geared toward ensuring integrity of files and their names, and in particular preventing filename manipulation attacks. In such attacks, a victim program expects a particular filename to have certain semantics. (E.g., a mail program may expect the file `/var/mail/root` to be the mail file for the super-user.) In the attack, the adversary creates a link by the same name in the filesystem, pointing to another file (e.g., `/var/mail/root -> /etc/passwd`), thereby "tricking" the victim program into accessing an unexpected file. (In the mail example, such a link may cause a naive mail program to write incoming email into the system's password file.) Such attacks were quite common in UNIX systems of old.

Since creating links to files often takes lower permissions than accessing these files, this form of attack sometimes allows an attacker to leverage the permissions of a privileged victim program to read or write files that the attacker cannot access on its own.

Our implementation is an abstraction of a safe file handling primitive which is presented in Section 6. The SimpFS interfaces are designed to tightly bind files with their names: files can be accessed *only* via the names they were created with, which means that filename manipulation attacks are impossible in our model. Our proof implies in particular that it indeed eliminates these filename manipulation attacks.

SimpFS offers a simple interface that captures enough filesystem primitives for application developers to build meaningful applications. The simplicity of SimpFS is due to its very narrow interface (only four commands) and the fact that *it does not have directories*. We argue that the murky relation between files and their names in plain POSIX systems stem to a large extent from the fact that pathnames consist of many directories, each with its own permissions, which are combined in a non-obvious manner to yield the effective permissions for the entire name. In contrast, a filename in SimpFS is just a single entity with explicitly specified permissions. Thus SimpFS provide applications with radically simplified semantics, making it easier to use the filesystem without falling into traps. At the same time, we

argue that the vast majority of contemporary applications in POSIX systems *do not really need directories*, and can be implemented over the simple SimpFS interface without loss of functionality.

5.1 Related Work

Triggered by Joshi and Holzmann's mini-challenge [JH07], there is a lot of recent work on formalization and verifications of file systems. Most notably, Freitas et al [FWF09] specify and prove a POSIX file store in Z/Eves. This body of work focuses mostly on the correctness aspects and does not address in depth the security and access control aspects of filesystems. In the broader perspective of (secure) operating systems, there is a long history of formalization and verification, from PSOS [NF03] to the recent seL4 [KEH+09]. While they make considerable progress toward high-assurance OS, these works are not based on frameworks that allow easy composition of components to form larger systems. Additionally, the focus in many of these works is on mandatory access control whereas we cover a discretionary control. (We stress that although our model addresses integrity concerns, these are very different from the Biba integrity model [Bib77].)

An abstract model of another large standard systems, the browser, suitable for proofs of cryptographic protocols exists in [GPS05]. It includes a model of information-flow properties under attack. However, the federated identity protocols built on top of it have only been proven secure with respect to specific security properties, not in a real-world / ideal-world setting [GPS05].

Protocol Composition Logic (PCL) [DDMR07a] is a comparable general approach on reasoning about (cryptographic) network protocols in a composable fashion. Recently, PCL was applied to analyze systems [DFGK09], more specifically integrity properties provided by TPM. The symbolic and axiomatic nature of PCL leads to a more axiomatic specification of security rather than the declarative form in UC. Furthermore, the composition theorems in PCL are weaker than in the UC framework.

A noteworthy contribution to secure composition of large software systems is the CHATS project [Neu], that identifies architectural principles to guide the structuring and decomposition of trustworthy systems. That work is largely orthogonal to ours, as it does not focus on formal modeling or proofs.

There have been many more attempts to leverage well-established formalisms such as logic, typing or process calculi to model composability of certain system security properties, e.g., McLean [McL96] for non-interference properties or Bengtson et al [BBF+11] for cryptographic protocols and access control mechanisms. Many of them provide tool support; but they do not provide the same composition guarantees as in the UC framework.

5.2 Conventions for Software systems

The UC Framework was largely defined to precisely capture cryptographic protocols and primitives which are very mathematically exact. The formal description of the artifacts is

done in terms of probabilistic interactive Turing machines (ITMs) which are more amenable to precise analysis. Using this to model OS services requires extensions to model many features such as preemptive multitasking, processes, kernel functions etc. Instead of writing down an entire new set of definitions in terms of probabilistic Turing machines, we briefly describe, at a very high level, some technicalities that must be resolved when attempting to apply the UC framework to software system, and the conventions that we use to address them. The "entities of interest" in our work are processes, which differ somewhat from the interactive Turing machines in common cryptographic models. One aspect relates to side-channels: whereas an ITM can only influence other ITMs by sending messages, a process shares some physical resources with other processes on the same machine, so it could influence them via side channels such as timing and concurrency. In this work we ignore that aspect, i.e. we do not have any side channels in our formal model. (This does not matter for our current SimpFS model, since we do not model any secrecy requirements.) We thus just let the adversary learn "whatever it needs," so it has no use for side channels.

A more important difference is preemptive multitasking: common cryptographic models postulate a sequential scheduling model, where an active ITM keeps the control until it sends a message, at which point the recipient becomes active. On the other hand, processes in contemporary OSes can be made to yield control involuntarily. Resolving this discrepancy is not as hard as it may seem, since (side-channels aside) an active entity has no effect on its surroundings until it sends a message, which means that influencing the surroundings only comes with losing the control. We use the standard sequential scheduling of the UC framework, but ensure that the adversary gets the control after every message is sent, and can decide when this message will be delivered. (This is somewhat similar to the "buffer scheduler" from [BPW07].) Hence the adversary in our formal model is able to simulate the actions that would have happened in the actual deployed system, delay delivery messages until the simulation arrives at the point where they were delivered. We thus argue that the formal adversary in our model is able to induce any behavior that can happen in the actual deployed system.

Another difference is that some processing in real systems is done not by the processes themselves, but by the kernel on their behalf. Hence also in our model we postulate the existence of a "kernel component" that can do things on behalf of processes. In our filesystem example, this kernel component is only responsible for maintaining the process privileges: Whenever a process calls a filesystem function, the kernel adds the process-id and roles of the calling process to the list of arguments, and forwards everything to the filesystem. (The kernel component gets these roles from the environment.) We note that although we do not use it in our filesystem example, in general we could have several such "kernel components" in a system, representing several physical machines.

5.3 SimpFS: A Simple Idealized File-System

This section describes SimpFS, our simple filesystem model. SimpFS has a minimalistic interface with simple semantics, having only basic primitives to create, read, write and delete

files. Still, we believe that the this file-system functionality is sufficient for most applications. (Other aspects — such as locking — can be implemented on top of our interface.) The SimpFS model includes file write permissions, hence capturing properties of *filesystem integrity*. We currently do not model read permissions, but we expect that this work can be extended to include read permissions without too much change.

An important feature of SimpFS is that *it does not have any directories, only files and their names.* As we mention in the introduction, we believe that directories have "inherently cumbersome semantics", hence decided to do away with them in order to keep the semantics as simple as possible. We stress that the model supports names that include '/' (so applications can still store their temporary data in files with names that begin with "`/tmp/`"). But a name such as "`/a/b/foo`" is viewed as just one entity, and its existence does not imply the existence of an object with name "`/a/b`." Of course, our *implementation* over POSIX still interprets '/' as a directory separator, and name creation induces the right associations between names and paths, in spite of symlinks, adversarial write permissions etc. While directories are a useful and convenient way to manage and organize systems, we argue that directory permissions are very rarely needed in applications (if ever), and most applications can therefore directly use the SimpFS interface.

A key security property of SimpFS is that it rules out filename manipulation attacks. Our focus on this property is motivated by the large number of privilege escalation attacks due to unsafe pathname resolution that were discovered in POSIX systems over the years. A classical example of this type of attacks is local mail delivery, where `/var/mail` may be world-writable, allowing an adversary to create a link from `/var/mail/root` to (say) `/etc/passwd`, thereby "tricking" a naive mail-delivery program (running as `root`) to write the content of incoming mail into `/etc/passwd`. Such attacks arise due to the opaque mapping of names to files in POSIX. SimpFS features a very tight binding between files and their names: a file can be manipulated *only* with the names it was created with.

We describe an implementation of SimpFS over contemporary POSIX filesystems and *rigorously prove* that this implementation realizes SimpFS, using the UC framework. The proof implies that processes that use our implementation will be protected against pathname manipulation attacks such as above even if adversarial processes use the same POSIX filesystem in arbitrary ways.

5.3.1 A formal model of SimpFS

SimpFS consists of files and their names. A newly created file is given some names, and thereafter the file can be accessed by any of these names. Existing names can be deleted, but one cannot add names to existing files. When deleting names, a file can end up with zero names, in which case it is not reachable anymore so we can consider it as deleted. We associate permissions with both the file names and the files themselves:

- Every file has a list of roles that can write in it, called the *Writers* list. A process can write to a file if it holds a role in the Writers list of the file.

```
CreateFile(Writers, Manipulators, Names, pid, Roles)
{
  // Allow the adversary to fail the operation and decide the error code
  var retCode = AdversaryAction("CreateFile",Writers,Manipulators,Names,pid,Roles);
  if (retCode != OKAY) return retCode;

  var codes[] = empty;    // a local list of return codes, one per name
  var f = index of next available entry in the files[] array;
  files[f].data=empty, files[f].Writers=Writers;

  // Allow the adversary to decide whether to create each name
  for each fName in Names {
    var code = AdversaryAction("CreateOneName", fName);
    if (code!=OKAY) codes[i]=code;
    else {
      if (names[fName] already exists) codes[i] = FILE_EXISTS;
      else {
        names[fName].file=f, names[fName].Manipulators=Manipulators;
        codes[i]=OKAY;
  } } }
  call AdversaryAction("Done CreateFile") and then return codes;
}

DeleteName(fName, pid, Roles)
{
  // Allow the adversary to fail the operation and decide the error code
  var retCode = AdversaryAction("DeleteName",fName,pid,Roles);
  if (retCode != OKAY) return retCode;

  if (names[fName] does not exist) return FILE_DOESNT_EXIST;
  if (Roles intersect names[fName].Manipulators = emptyset) return NO_PERMISSION;

  delete names[fName]; // Note: no point deleting the file, even if not reachable
  call AdversaryAction("Done DeleteName") and then return OKAY;
}

Write(fName, atAddr, data, pid, Roles)
{
  // Allow the adversary to fail the operation
  var retCode = AdversaryAction("OpenWrite",fName,pid,Roles);
  if (retCode != OKAY) return retCode;

  if (names[fName] does not exist)  return FILE_DOESNT_EXIST;
  var f = names[fName].file;        // f serves as a "handle" to the file
  if (Roles intersect files[f].Writers = emptyset) return NO_PERMISSION;

  var numBytes = AdversaryAction("Write",fName,atAddr,data,pid,Roles);
  if (numBytes < length(data)) truncate data to numBytes bytes;   // only partial write

  var nBytes = length(data);
  if (atAddr < 0) atAddr = length(files[f].data);    // append
  else if (atAddr > length(files[f].data)) {
    prepend (atAddr-length(files[f].data)) zero bytes to data;
    atAddr = length(files[f].data);
  }
  write data to files[f].data starting at position atAddr;
  call AdversaryAction("Done Write") and then return [OKAY,nBytes];
}
```

Figure 1: SimpFS commands.

```
Read(fName, fromAddr, nBytes, pid, Roles)
{
  // Allow the adversary to fail the operation or read less bytes
  var [retCode,numBytes] = AdversaryAction("Read",fName,fromAddr,nBytes,pid,Roles);
  if (retCode != OKAY)  return retCode;
  if (numBytes < nBytes) nBytes = numBytes;

  if (names[fName] does not exist) return FILE_DOESNT_EXIST;
  var f = names[fName].file;

  if (fromAddr < 0) fromAddr = 0;
  else if (fromAddr > length(files[f].data)) {
    fromAddr = length(files[f].data);
    nBytes = 0;
  }
  if (nBytes < 0)   // read to end-of-file
    nBytes = length(files[f].data) - fromAddr;

  data = content of files[f].data from fromAddr for nBytes;

  call AdversaryAction("Done Read") and then return [OKAY,nBytes,data];
}
```

Figure 2: Other SimpFS commands.

- File names have a set of *Manipulators*, listing all the roles that have permission to delete that name.

In the current version we do not have read permissions, which means that SimpFS allows every process to read every file.

In more details, our ideal SimpFS maintains an array of files and an associative array of names: `files[]` is an array of files (indexed by integers). Each entry is a file, consisting of an array of bytes (i.e., a data blob) and a list of roles (specifying the Writers of this file). `names[]` is an associative array (indexed by strings). We refer to the index of an entry as a file-name, and each entry consists of a pointer to a file (i.e., an integer) and a list of roles (specifying the Manipulators of this name). The interface below constrains the Manipulator lists, making sure that all the names of the same file have the same set of Manipulators. (This choice is not very important, it is done mostly to simplify the presentation.)

In the initial state, the file-system is empty, with no files and no names (i.e., both arrays are empty). There are only four operations that are supported in SimpFS: `CreateFile` creates a new file with some names, `DeleteName` deletes an existing name, `Read` reads data from a file (specified by some name), and `Write` writes data to a file (specified by some name).

The semantics of these operations is described by the pseudo-code in Figure 1 and Figure 2. As is the case with every formal UC functionality, the pseudo-code includes not only the intended functionality as seen by the legitimate users of the system, but also all the interfaces that an adversary can utilize to attack it. This is codified by an `AdversaryAction` call, in which SimpFS "leaks" to the adversary the details of its operation, and also lets the adversary influence these operations.

A key feature of SimpFS is that a file can be accessed *only* using one of the names that were specified when the file was created, thus eliminating filename-manipulation attacks such as described above. Hence proving that an implementation realizes SimpFS implies in particular that such attacks cannot be successfully mounted against the implementation.

We make no liveness guarantees in SimpFS, so at the beginning of every operation the adversary is given the option to abort the operation and determine the error code. (This does not mean that an implementation of SimpFS cannot ensure some liveness properties, but it means that a proof that an implementation realizes SimpFS carries no such guarantees within itself.)

The pseudo-code includes with every call also the process-id and permissions (Roles) of the caller, which in our system model are filled by the kernel component, cf. Figure 3.

(Formally there is also an implicit "invocation id" for each call of one of the four main operations, allowing SimpFS to handle messages received from the ideal-world adversary for different invocations.) Note also that the `AdversaryAction` at the beginning and end of every operation comply with our convention that the adversary gets the control before any message is delivered. Finally, we note that all the variables in the code in Figure 1 are local to that invocation, except for the global `files[]` and `names[]`.

Process corruption. Following the standard conventions of the UC framework, SimpFS has a special procedure to handle the case where the adversary corrupts a process. For our purposes it is more convenient to let the environment decide when a process is corrupted (as opposed to the adversary, which is the more common convention in UC-model works). When the environment corrupts a process, this process makes a call `IamCorrupted(pid,Roles)`, to inform SimpFS that "it belongs to the adversary" now. SimpFS informs the adversary of this call, and it remembers that this process and all its roles are now bad. Thereafter, the adversary is allowed to make all the usual calls to SimpFS (`CreateFile`,`DeleteName`,`Read`, `Write`) on behalf of that process. SimpFS will process these calls just as if it was the corrupted process that made the call, but will return the result to the adversary rather than to the environment.

Every call from the corrupted process (not via the adversary) will be routed directly to the adversary, and the adversary can always instruct SimpFS to send anything to the corrupted process (which will then be forwarded to the environment). Also, if the roles of the corrupted players change then the kernel component will notify SimpFS of this change. SimpFS will add any new role that a corrupted process acquires to its list of bad roles, but *it will not remove any roles from that list*, even if the corrupted process loses some of its roles. (This last aspect represents the fact that the corrupted process may already have used this role to introduce artifacts into the filesystem, that will remain even after the process no longer has this role.)

Atomicity of the SimpFS operations. The operations `DeleteName` and `Read` are atomic, whereas `CreateFile` and `Write` are not: In `DeleteName` and `Read`, once the adversary allows the operation to go through (by returning `OKAY`), SimpFS holds onto the control-flow throughout the name lookup and the operation itself, and only then it yields

control back to the adversary.

In `Write`, on the other hand, the control is returned to the adversary after the file lookup (via the call `AdversaryAction("Write", ...)`), and only then is the operation carried out. Similarly in `CreateFile`, the adversary gets the control before the creation of any name. This choice was made so that we would be able to realize SimpFS over the POSIX interface that requires to open the file and then write in it. The real-world read can be made atomic by checking after the fact that the file did not change since it was opened, but for write such a check is meaningless since the file was already written. (See also the attack in Section 5.4.5.2 for another reason for the check after read.)

Mapping Unix permissions to roles. The interfaces of SimpFS above are defined with "generic roles" that encode permissions, with access control being a simple role inclusion. Our implementation over POSIX, of course, uses userids and groups, which are particular types of roles. The mapping is quite straightforward, roughly there is a different role for each userid and group in the system, and a process gets the role corresponding to its effective-uid and all the roles corresponding to its groups. There is also one role for "others", that every process has. Some care must be taken since POSIX permissions do not exactly follow role inclusion. (For example, if a file is not owner-readable then the owner cannot read it, even if the file is readable by "others".) Adjusting the mapping to this technicality is quite straightforward, and is omitted here.

5.4 Implementing SimpFS over POSIX

We describe simpfs, which is a concrete implementation of the SimpFS functionality over the POSIX filesystem interface [IEE08]. The presentation below focuses on a user-space implementation, where each simpfs operation runs with the effective uid of its caller, but we point out that the same procedures can also be implemented in the kernel. (See Figures 3 and 4 for illustrations of the system model in both cases.)

Our implementation relies on the "safe pathname resolution", described in more detail in Section 6, that protects processes from opening adversarial links. While resolving paths this procedure ensures that an adversary can not manipulate the resolution to result in opening unintended components. In simpfs, very roughly speaking, each operation consists of first using that procedure to open the corresponding file and then performing the actual operation.

Before describing this implementation, we first introduce concepts that are used in the rest of the section and describe some assumptions that we make on the POSIX filesystems underlying our implementation. Then in Section 5.4.2 we describe the safeDirOpen procedure, which is the heart of our implementation, and then in Section 5.4.3 we describe the rest of the implementation.

5.4.1 Concepts and Properties of POSIX

We assume that the reader is familiar with basic concepts of POSIX such as directories, pathnames, users and groups, hardlinks and symlinks, etc.

Definition 1 (Pathname Manipulators) *Let* `/dir1/.../dirn/foo` *be an absolute pathname. The* manipulators *of this pathname are all the roles (users and groups) that own, or have write permissions in, any directory visited during the resolution of this pathname.*

Note that the definition applies even when a pathname does not resolve, and that `root` is a manipulator of every pathname.

Definition 2 (Safe Names) *A pathname is* system safe *if its only manipulator is* `root`*. A pathname is* safe for *U (where U is a user-id) if its only manipulators are* `root` *and U. Otherwise, the pathname is unsafe for U.*

For example, in a typical UNIX system the pathname `/etc/passwd` is system safe, the pathname `/home/joe/mbox` is safe for user `joe`, and the pathname `/var/spool/mail/jane` is unsafe for everyone (as `/var/spool/mail` may be world- or group-writable).

Definition 3 (Simple Pathnames) *A pathname is* simple *if it is an absolute path that resolves to a regular file, its elements are only hard links (i.e., not symbolic links), no elements are named '.' or '..', and the pathname contains no repeated slashes '//'.*

Assumptions. We now list some properties that we assume on the underlying POSIX system, and use in our proof of security. Most of these assumptions are justified either by the fact that they are part of the POSIX specification itself, or by the fact that many contemporary POSIX filesystems seem to satisfy them.

Assumption 1 *The underlying filesystem does not contain multiple mount points to the same filesystem, and each directory has only one parent (i.e., one hard link with a name other than '.' or '..').*

Justification. Assumption 1 is justified by the fact that nearly all contemporary POSIX implementations either do not allow processes to create additional hard links to directories (e.g., FreeBSD, Linux) or restrict this operation to the super-user (e.g., Solaris, HP-UX). A notable exception is MacOS.

We observe that given Assumption 1, for every reachable hard link to a regular file there is a unique simple name that ends with that hard link. Moreover a resolution of any absolute name that ends with that hard link will visit all the directories in this unique simple pathname.

Assumption 2 (Permissions) ***1.*** *If an operation by a process affects the content of a file, then the process must have write permission for that file.* ***2.*** *Let P be an absolute pathname. If an operation by a process affects the resolution of P or changes the permissions or ownership of any of the directories visited during its resolution, then that process must have a role which is a manipulator of P.*

Justification. The only operations that affect pathname resolution are creating, removing, or renaming pathname components, and they all require write permission in the containing directory. Also, note that only the owner of a directory (or `root`) can change the permissions of that directory, and in most systems only `root` can change ownership.

Corollary 3 *Let P be some pathname, denote by $\mathcal{M}(P)$ the set of manipulators for P (user-ids and groups), and let $\mathcal{B}$ be a set of roles such that $\mathcal{M}(P) \cap \mathcal{B} \neq \emptyset$. Then changing the manipulator set for P so that $\mathcal{M}(P) \cap \mathcal{B} = \emptyset$ requires an operation by a process with some role outside of $\mathcal{B}$.*

Proof: The only operations that change the manipulator-set of a pathname are changing the permissions or ownership of some visited directory, or moving, renaming, or removing some visited directory, symlink, or the last hardlink.

Denote by `op` the first system call after which the manipulator-set of of P is disjoint from $\mathcal{B}$. Denote by $\mathcal{M}'(P)$, $\mathcal{M}''(P)$ the manipulator set of P just before and just after the system call `op`, respectively, so $\mathcal{M}'(P) \cap \mathcal{B} \neq \mathcal{M}''(P) \cap \mathcal{B} = \emptyset$. Since `op` changes the manipulator set of P, it must have succeeded, hence the calling process must have had some role R^* with sufficient privileges for performing `op`.

Assume toward contradiction that the calling process has only roles in $\mathcal{B}$, and thus $R^* \in \mathcal{B}$. Since R^* has sufficient privileges for one of the manipulator-changing operations then by Assumption 2 $R^* \in \mathcal{M}'(P)$. We now have three cases: either `op` is `chown` (so R^* is `root` hence it remains a manipulator), or `op` is `chmod` (so R^* is the owner of the directory so it remains the owner), or `op` is any other manipulator-changing operation so R^* is a writer in the containing directory and it remains so after the operation. In each case R^* remains a manipulator, $R^* \in \mathcal{M}''(P) \cap \mathcal{B}$, hence $\mathcal{M}''(P) \cap \mathcal{B} \neq \emptyset$. □

Assumption 4 *The hardlink to a directory in its parent directory can only be removed when the child directory is empty. Moreover, after the hardlink is removed from the parent directory, no further entries can be created in the child directory, even if some process still holds a handle to it.*

Justification. The last part of Assumption 4 is justified by the fact that `rmdir` implementations remove the entries '`.`' and '`..`' from the child directory before removing the hard link in the parent directory, and no new entries can be created in directories without '`.`' and '`..`'.

Corollary 5 *If a system call for creating an entry in a directory returns successfully, then the hard link for this directory in its parent directory could not have been removed before that system call, or removed after the call but before the newly-created entry is removed.*

5.4.2 The safeDirOpen procedure

Underlying our simpfs implementation is a procedure for *safe name resolution*. Our procedure, safeDirOpen, takes an absolute pathname, resolves it "in a safe manner" and returns a handle to the directory containing the final hard link to the actual file, the name of that hard link, and additional information as discussed below. The top-level operations of simpfs first call safeDirOpen and then perform the requested operation on the final hard link.

safeDirOpen resolves a pathname one atom at a time, each time opening the next atom (or reading it, if it is a symlink), while keeping track of the owners and writers of the visited directories. (Below we identify the time that a directory was visited as the time when it was opened, and the time that a symlink was visited with the time that it was read.)

The procedure can be in one of three states: system-safe, safe-for-uid, or unsafe. When invoked (by a process with effective uid U), the procedure begins in a system-safe state, switching to safe-for-uid state upon visiting a directory where U is an owner or writer, and switching to unsafe state upon visiting a directory with any writer or owner other than `root` or U. Once in unsafe state it stays in that state for the duration of the current name resolution. Likewise, there is no transition from safe-for-uid to the system-safe state.

When safeDirOpen enters the unsafe state, it does not follow symlinks for the remainder of the current name resolution. Also, for technical reasons the procedure never accepts pathnames that contain multiple slashes '`//`' or have components named '`.`' or '`..`', and it refuses to visit any directory whose name begins with the special prefix `_SimpFS_ephemeral_`. In any of these cases, the procedure returns an error code.

Once safeDirOpen arrives at the final atom (and verifies that it is indeed the final atom and not a symlink), it ends successfully, returning a handle to the directory containing this last hard link, as well as the name of the hard link. In addition, safeDirOpen returns its current state (system-safe, safe-for-uid, or unsafe), the set of owners and writers of the directories that it visited, and an array of (handle,name) pairs, containing handles to all visited directories, and the names that were looked-up in those directories. (These names could belong to either a directory, a symlink, or the final hard link.)

Upon failure, safeDirOpen returns an error code, a handle to the last directory pathname component that was successfully resolved, the state (system-safe, etc.) and manipulators of that directory, and the unresolved remainder of the pathname. For example, when called to resolve `/a/b/c`, if it encountered an error after visiting `/a` but before visiting `/a/b`, then it will return a handle to directory `/a`, the state and manipulators of `/a`, and the remainder of the pathname argument "`b/c`". (Note that this will be the return value even if `/a/b` happens to be a symlink and the procedure visited more directories after `/a`, but could not completely resolve `/a/b`.)

5.4.3 Implementing the simpfs commands

createFile(`Writers,Manipulators,Names`). When called by a process with effective-uid U, the procedure begins by checking that U belongs to the set of manipulators specified by the `Manipulators` parameter. Then it creates a new file with an ephemeral name that begins

with the special prefix `_SimpFS_ephemeral_`. This ephemeral name is created so that it is safe for U, thus ensuring that no other users can remove or rename it.See Section 5.4.5.4 for a short discussion of this point.

Now createFile attempts to set the write permissions of the new file as specified in the `Writers` parameter. If this is successful, it proceeds to create the names, one at a time, by calling the subroutine createOneName for each name in `Names`. After all the calls to createOneName, the procedure createFile removes the ephemeral name that it created for the new file, and returns the vector of return codes that it received from all the calls to createOneName.

The subroutine createOneName(`fName`) begins by checking that the new name is an absolute name, and that it does not contain '`//`' or elements named '`.`' or '`..`', or elements that begin with `_SimpFS_ephemeral_`. Then it calls safeDirOpen(`fName`) thus obtaining a handle to the last successfully resolved directory on this pathname and the corresponding set of manipulators. If all the directories were resolved successfully, then createOneName checks that the set of manipulators equals the `Manipulators` parameter, and aborts if they differ.

If some directories were not resolved, createOneName verifies that the manipulator set of the prefix is not too large (i.e., it must be contained in the `Manipulators` parameter), aborting otherwise. Then createOneName attempts to create the remaining directories, one at the time, initially creating each one so that it is only writable by owner U with an ephemeral name that begins with `_SimpFS_ephemeral_`. Upon success, it tries to set the write permissions of the last directory so that the resulting set of manipulators will match the `Manipulators` parameter. Then it goes over all the newly created directories, top to bottom, renaming each one to the name that it is supposed to have according to `fName`.

Once all the directories exist and have the right set of manipulators and the right names, the procedure createOneName makes a `linkat` system call to create a hard link in the last directory, pointing to the new file. createOneName then returns whatever code was returned from the `linkat` system call.

If any operation fails, then createOneName attempts to clean-up after itself, trying to remove all the directories that still have names that begin with `_SimpFS_ephemeral_`. However, after a directory was renamed to its "permanent name", createOneName does not remove it.

In the proof of security in Section 5.5 we rely on the following properties of our implementation of createFile:

- The initial ephemeral name for the new file is safe for the effective-uid of the calling process.
- The procedure never creates symlinks, only directories and hard links.
- The procedure only changes permissions and/or removes pathname components if these components begin with the special prefix `_SimpFS_ephemeral_`.
- A name `fName` is created if and only if the `linkat` system call at the end of the subroutine createOneName(`fName`) is successful.

deleteName(fName). When called with effective-uid U, deleteName calls safeDirOpen(fName) and aborts if that function fails. Else deleteName has an array of pairs (handle,name), and the state with which safeDirOpen arrived at the final directory (system-safe, safe-for-uid, or unsafe). If the state is not system-safe, then deleteName checks that the final directory is either world-writable, or owner-writable and owned by U, and it aborts otherwise.This check is intended to protect against privilege-escalation attacks on `setgid` programs, cf. Section 5.4.5.1. Also, if the state is unsafe then deleteName checks that the file that the hard link points to has only a single hard link, aborting otherwise.

Then deleteName attempts to delete the final hard link, followed by attempts to delete the directories higher-up on the path. deleteName returns when any system call to remove a name fails, or when any of these names resolves to a symlink, or when it is done deleting all the names in the array. The return code from deleteName is whatever was returned from the first `unlink` system call (i.e., the one that deleted the hard link at the end of `fName`).

We note that barring a race condition, this implementation of deleteName does not delete symlinks. In the proof in Section 5.5 we show that the only cases where these race conditions are possible are when the adversary already has permissions to delete these symlinks by itself.

read(fName,...). When called with effective-uid U, read calls safeDirOpen(fName) to get a handle for the final directory, the name of the hard link pointing to the actual file, and the state at which it arrived in this last directory: system-safe, safe-for-uid, or unsafe. Then read uses `openat`, `lstatat` and `fstat` to open the file and verify that it is still the same file (and not a symlink). In addition, if the state is not system-safe, then read checks that the file is either world-readable, or owner-readable and owned by U, and it aborts otherwise. Also, if the state is unsafe then read checks that the file has only a single hard link, aborting otherwise.

Then the procedure uses the `read` system call to read the file, and before closing the file it makes yet another `lstatat` system call to check that the hard link still points to the same inode as it did when it was `open`ed. (See Section 5.4.5.2 for the reason for this last test.) If all these checks pass, then read returns the result from the `read` system call.

write(fName,...). The procedure write is almost identical to read except that it adds a write-permission check on the actual file, and it does not do the final check after writing to verify that the hard link still points to the same inode. (Indeed, such check is useless since the file was already written to.)

5.4.4 Consistency properties of the implementation

In the proof of security in Section 5.5, it is important to consider what changes may happen in the filesystem between the time that the safeDirOpen pathname resolver visits some directory and the time that the procedure that called safeDirOpen returns. An important technical observation is that if the procedure that called safeDirOpen was successful then none of those visited directories could have been removed during this time.

Lemma 6 *Consider an execution of one of the procedures createOneName, deleteName, read, or write on argument fName, and assume that the procedure succeeds (i.e., does not return an*

error code). Assume further that no symlink that was read during name resolution was later deleted or renamed during the execution of this procedure, and no directory was renamed after it was `open`*ed by this procedure. Then also none of these directories was deleted after it was* `open`*ed and before the time that the procedure issued the system call (respectively,* `linkat`*,* `unlinkat` *or* `openat`*) for the final hard link in* `fName`*.*

Moreover, for the procedures createOneName*,* read*, and* write*, as long as no symlinks are deleted or renamed, no directories are renamed, and the final hard link in* `fName` *exists in its original containing directory, then also none of these directories is deleted even after the operation returns.*

Proof: Assume not, and consider the first directory that was deleted after it was `open`ed. There are two cases to consider: this directory was deleted either before or after name resolution visited the next pathname component (i.e., symlink read, directory or file `open`ed).

By Assumption 4, the directory could not have been deleted before the next component was accessed, else the subsequent access would have failed. But it also could not have been deleted after the next pathname component was visited, since the directory must have been non-empty: If the next pathname component is a symlink then this follows from our assumption that symlinks were not removed or renamed, if it is a directory then it follows from our assumption that directories were not renamed and the fact that we consider the first directory to be removed, and if it is the final hard link then it follows from our assumption that it still exists in its containing directory. □

Jumping ahead, we use Lemma 6 in the proof by noting that our SimpFS implementation never renames or removes symlinks, or renames directories, and hence no uncorrupted process will do any of these things. If in addition we know that no corrupted process has write permissions in any of the directories visited then also corrupted processes could not rename or remove symlinks or rename directories. Thus, we can apply Lemma 6 and conclude that all the directories stay put throughout the execution of createOneName, deleteName, read, or write.

5.4.5 Rationale and Discussion

Before proceeding to the formal proof of security, we discuss here some of the rationale for our implementation, including some specific attacks that the implementation was designed to foil.

5.4.5.1 Privilege-escalation attacks on `setgid` programs Our implementation of safeDirOpen only considers the effective-uid for the purpose of determining the safety of a directory, and thus we must consider the possibility of privilege-escalation attacks between processes with the same effective-uid. In contemporary UNIX systems, two processes with the same effective-uid can have different filesystem privileges only if one of them has a group-privilege that the other does not,(we ignore the `fsuid` of Linux here) as would happen when one of these processes runs a `setgid` program.

To see the problem, consider two processes running with effective-uid of `joe`, one having the additional group privilege of `mail` while the other is compromised by an attacker (e.g.,

due to a buffer-overflow vulnerability). Ideally, we would like to argue that files which have read/write permissions for the `mail` group (but not user `joe`) are still protected against the compromised process.

Assume that the non-compromised process with `mail` group privileges needs to delete a file `/home/joe/dir/foo`. The compromised process can create a symlink `/home/joe/dir -> /var/mail`, "tricking" the other process into deleting `/var/mail/foo` (assuming that `/var/mail/` is writable by group `mail`). Embedding this attack in our formal model, we have a name `/var/mail/foo` for which `joe` is *not a manipulator*, and a good process that attempts to delete an unrelated name `/home/joe/dir/foo`, and yet by some action of a compromised process with `joe` privileges, this results in the deletion of `/var/mail/foo`.

We fix this problem by adding a check to the operations deleteName, read, and write, aborting if the name is not system-safe and group privileges are needed to perform the operation. Very roughly, this defense works because it prevents the use of group privileges after following symlinks that were created by non-`root` processes. (We note that we do not need this extra precaution in createOneName. This is because the SimpFS functionality restricts deletion of existing names, but puts no restrictions on the creation of names that do not exist.)

5.4.5.2 An attack on open-then-read programs To understand the need for another check of the final hard link after a `read` system call in a read operation, we describe the following potential attack: Consider the three programs `sshd` that needs to read the file `/etc/passwd`, `passwd` that replaces the file `/etc/passwd` by a new file upon successful edit, and the MTA local delivery that needs to write into `/var/mail/root`. The `passwd` program runs with `root` privileges, because it is a `setuid`-root program, and the MTA local delivery runs with `root` privileges in order to append to the `/var/mail/root` mailbox file. Also, assume that the directory `/var/mail` is world writable and that initially `/var/mail/root` does not exist. The attack consists of the following sequence of steps:

1. The attacker creates a *hard link* `/var/mail/root`, pointing to `/etc/passwd`.

2. The attacker opens a new `ssh` connection, causing `sshd` to `open` the file `/etc/password` for read. (Note that since `/etc/passwd` is a system-safe name, the `open` will succeed even if there are multiple hard links.) At this point the attack relies on the `sshd` process to be switched out and remain inactive until Step 5 below.

3. The attacker then uses the `passwd` command to change its password, thereby causing the old `/etc/passwd` file to be replaced by a new file. (Note that the hard link `/var/mail/root` is now the only hard link still pointing to the old `/etc/passwd` file, and that the `sshd` process still holds a handle to that file.)

4. The attacker sends email to `root@localhost`, causing the MTA local delivery to append the content of that message to `/var/mail/root`.

5. The `sshd` process is now switched in again and reads from its handle to the old `/etc/passwd` file, thereby reading also the data that was written there by the MTA delivery agent.

To thwart this attack, we added the `lstatat` check between reading and closing the file, verifying that the hard link still points to the same file. We stress that it is possible to switch the link back and forth to foil this extra test, but it is sufficient for the purpose of our simpfs implementation. Non-adversarial processes will never attempt such a back-and-forth switch, and adversarial processes either do not have the privileges needed to foil the test, or else they have sufficient privileges to manipulate the file directly. (Our proof relies on this extra test in the analysis of the read operation).

5.4.5.3 Our treatment of symbolic links Our proof of security in the full version Section 5.5 relies in places on the assumption that good processes do not create symlinks. This is consistent with our simpfs implementation (that indeed does not create symlinks), but it begs the question why we allow safeDirOpen to follow symlinks at all.

The reason is that the implementation of simpfs is useful also in situations where the filesystem includes non-adversarial symlinks. A close inspection of our proof shows that the arguments remain valid also in the presence of non-adversarial symlinks, as long as the files that have non-adversarial symlinks in their names remain static (i.e., they are not deleted, removed, or moved). It is even possible to modify the semantics of SimpFS to accommodate non-adversarial symlinks in a dynamic filesystem, but the new semantics will not be as simple anymore.

5.4.5.4 Using the sticky bit Recall that the initial ephemeral name for a new file must be safe for the effective-uid of the calling process (denoted U). Such a name can perhaps be created in U's home directory, but not all uid's have one. A simple way of achieving the same result in contemporary UNIX systems is creating this ephemeral name in `/tmp`, relying on the fact that `/tmp` is owned by `root` and has the sticky bit on. This does not quite fit into our definition of "safe for U" (since `/tmp` is world-writable), but it suffices for the purpose of our proof of security. Specifically, what we need is to ensure that as long as the calling process holds a handle to the new file, only U or `root` can change the resolution of the ephemeral name.

5.5 Proof of Security

We next prove that our simpfs implementation realizes the SimpFS functionality over POSIX, given our assumptions from Section 5.4. The proof refers to a system model where simpfs is implemented in user-level code and relies on an incorruptible kernel component that handles process permissions; see Figure 3. Essentially the same proof shows that the simpfs procedures realize the SimpFS functionality when implemented in the kernel (in which case permissions are handled by the environment, cf. Figure 4).

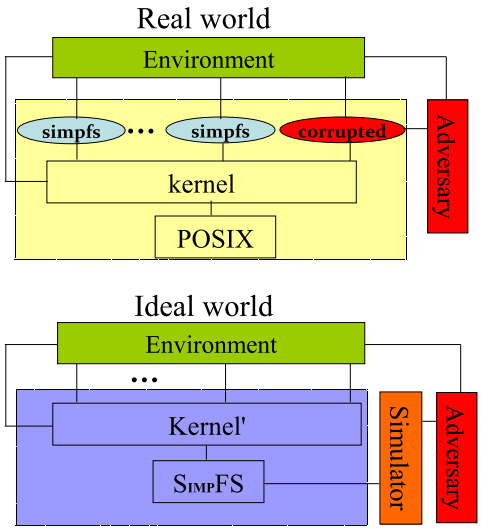


Figure 3: The real and ideal worlds for a user-level implementation of simpfs. The kernel components that keep track of privileges are formally considered to be parts of the implementation and the ideal functionality.

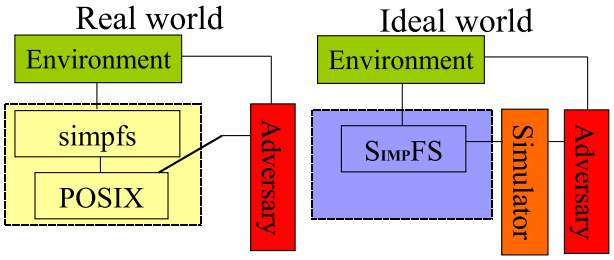


Figure 4: The real and ideal worlds for a kernel implementation of simpfs. In this setting, process privileges are handled by the environment.

Theorem 1 *Our simpfs implementation realizes the* SimpFS *functionality over the POSIX interface, provided that the underlying POSIX system satisfies Assumptions 1 through 4.*

To prove Theorem 1 we show that there exists an ideal-world simulator $\mathcal{S}$ such that for every real-world adversary $\mathcal{A}$, no environment $\mathcal{Z}$ can distinguish the behavior of the real world with $\mathcal{A}$ from that of the ideal world with $\mathcal{S}$ and $\mathcal{A}$. We first define a few concepts that will be important in the proof, then define the simulator $\mathcal{S}$, and finally prove the indistinguishability.

5.5.1 Useful Concepts

The simulated real world. As usual, our simulator $\mathcal{S}$ interacts with the adversary $\mathcal{A}$, and it needs to simulate a complete picture of the real world as would be seen by this $\mathcal{A}$ and the environment $\mathcal{Z}$. Note that $\mathcal{S}$ knows all the calls made by $\mathcal{A}$ to the underlying POSIX system. Also, the simulator knows the details of all the calls made by the legitimate players to the SimpFS functionality (by virtue of the `AdversaryAction` calls made by SimpFS). Hence $\mathcal{S}$ can simulate the corresponding real-world implementation for these calls, keeping a complete picture of the real-world POSIX system as it would exist in the real world at any point in time. Below we call this POSIX system that the simulator keeps the *simulated real world*.

Bad roles. Recall that the association between processes and roles (such as userid and groups) is *not one-to-one*. This raises the possibility that some roles are held by both corrupted and uncorrupted processes at the same time, and similarly a process can have both "good" and "bad" roles.For example, we could have a corrupted process with userid `jack` and group `users` and an uncorrupted process with userid `jane` and group `users`, so the role corresponding to group `users` is held by both an corrupted process and an uncorrupted one. Also the uncorrupted process holds both a "good" role (`jane`) and a "bad" role (`users`). To handle these cases we introduce the following definition.

Definition 4 (Bad roles) *At any point in a run of the system, the set $\mathcal{B}$ of bad roles contains all the roles that were held by a corrupt process since the start of this run. The other roles are called good roles.*

Clearly, the set $\mathcal{B}$ is monotonically growing throughout the run of the system. The simulator can make calls to SimpFS using any role in $\mathcal{B}$, as per our process corruption interface.

Protected names and files. Throughout the simulation, some of the names in the simulated real world also exist in the SimpFS functionality, while the others exist for the most part only "in the simulator's head." Intuitively, the former are the *protected* names while the latter are *unprotected*. The formal notions of protected names (and also files) are defined next.

Definition 5 (Protected Names) *An absolute pathname `fName` that resolves to a regular file in the simulated real world at a given point in time is* protected *if no bad role in $\mathcal{B}$ is*

a manipulator for `fName`*. Pathnames that resolve to regular files but are not protected are called* unprotected*.*

Definition 6 (Protected Files) *A file that exists in the simulated real world is* protected *if no bad role in* $\mathcal{B}$ *has permission to write in it. Otherwise it is* unprotected*.*

Unprotected names and files can exist only after some processes were corrupted. Also, a system-safe pathname is protected if and only if no `root` process was corrupted, and a pathname which is safe for U is protected if and only if no `root` or U processes were corrupted.

Note also that protected names must be created by uncorrupted processes, since no corrupted process has the permission to create them. This means that protected names can only be either the names that were specified as arguments to createFile, or the temporary names with special prefix `_SimpFS_ephemeral_` that are used inside the procedure createFile. Below we refer to the latter as *ephemeral*:

Definition 7 (Ephemeral Names) *A pathname in the simulated real world is defined to be ephemeral if any of the pathname components begins with the prefix* `_SimpFS_ephemeral_`*.*

5.5.2 The Simulator

The simulator's strategy is to keep in the SimpFS functionality only protected names, while it simulates the unprotected names internally. When a player tries to access such an unprotected name, the simulator temporarily creates a file with that name in SimpFS by making a `CreateFile` call on behalf of a corrupted process. The simulator then allows the main operation to succeed and return an answer, and then deletes that temporary name using a `DeleteName` call on behalf of the same corrupted process.

In a few more details, when the simulator is informed by SimpFS that some process invoked an operation (`CreateFile`, `DeleteName`, `Read`, `Write`), it simulates the corresponding procedure of the simpfs implementation (including any interleaving events). For `DeleteName` and `Read` it returns OKAY if the procedure succeeds, for `CreateFile` it returns the first OKAY once the temporary file is created and set with right permissions and then returns OKAY for each name for which the `link` system call succeeded. For `Write` it returns the first OKAY when the procedure `open`'s the file, and then again when the procedure successfully `write`'s to the file. In all cases, if the procedure in the simulated real world fails, the simulator returns the same error code.

For `DeleteName`, `Read`, and `Write`, before returning OKAY the simulator ensures that a file with the corresponding name exists in the SimpFS functionality. If this name is an unprotected, the simulator first creates a temporary file with this name in SimpFS and writes into it the content that it has in the simulated real world. The simulator also puts these temporary names in a list of names to-be-deleted, and deletes them from SimpFS as soon as it gets back the control. Similarly for `CreateFile`, if a successful createOneName

creates an unprotected name then the simulator puts that name on its to-be-deleted list and deletes it from SimpFS once it gets back the control.

When receiving a `Done Write` call from SimpFS, the simulator goes over all the protected file names, looking for names for which the content of the corresponding file in the simulated real world differs from that in the SimpFS functionality. If the file is *unprotected* (i.e., the simulator has permissions to write in it) then the simulator makes a `Write` call to set the content of the file in the SimpFS functionality to match that of the simulated real world.

Process corruption. When the simulator learns from SimpFS that a process is corrupted, it goes over all the file names that exist in SimpFS, and deletes each name that the newly corrupted process can delete from SimpFS, using a call on behalf of that process. The simulator also remembers that this process is now corrupted.

Modifications of files with protected names. When a corrupted process modifies the content of a file that has a protected name in the simulated real world, the simulator makes a `Write` call to the SimpFS functionality on behalf of the same process, setting the content of the corresponding file inside SimpFS to match that of the simulated real world.

5.5.3 Proof of correctness

We show that with the simulator defined above, the view of the environment in the ideal and real worlds is identical. As we noted above, it is sufficient to argue about the simulated real world vs. the SimpFS functionality. We now prove a sequence of lemmas relating the names that exist in the simulated real world to those that exist in the SimpFS functionality.

Lemma 7 *Every protected name that exists in the simulated real world is either ephemeral or also exists in the* SimpFS *functionality.*

Proof: Recall that protected names must be created by uncorrupted processes, since corrupted processes do not have permission to write in the directories containing them. As per our implementation, the only names of regular files that are created by uncorrupted processes are either the names that are specified as parameter in createFile or ephemeral names. As to the former, they are created via a successful `link` system call in createOneName, at which point the simulator returns OKAY to the SimpFS functionality, which in turn then creates the name (if it does not already exist). □

Below we say that the a particular link (hard or symbolic) that exists in the simulated real world *remains unchanged* during some time interval if it is not removed or renamed in its containing directory, and its permissions and ownership remained the same. A pathname remains unchanged if all the directories, links and filenames that are accessed during resolution of this pathname remain unchanged.

Lemma 8 *Every name* ***fName*** *that exists in the* SimpFS *functionality and no corrupted process has permission to delete it, also exists in the simulated real world and is protected. Moreover,* ***fName*** *remained unchanged since it was last created in the simulated real world.*

Proof: Fix any file name `fName` that satisfies the premise of the lemma. This cannot be temporary a name on the to-be-deleted list, since those can be deleted by corrupted processes. Thus the last time when it was created in SimpFS was after a successful `link` system call in createOneName, during a `CreateFile` call by an uncorrupted process. (Also `fName` is not ephemeral, since our implementation of createOneName does not create ephemeral names for regular files.)

Let $\mathcal{M}$ be the set of manipulators for `fName` in the SimpFS functionality, so by the premise of the lemma $\mathcal{M} \cap \mathcal{B} = \emptyset$. Also, $\mathcal{M}$ was the manipulator-set specified in the `CreateFile` call to SimpFS when `fName` was created. Recall now that the subroutine createOneName keeps track of all the owners/writers in all the directories that it visits, and only issues the final `link` system call if that set equals $\mathcal{M}$. Denote the directories visited during name resolution (in order) by $\texttt{dir}_1, \texttt{dir}_2, \ldots, \texttt{dir}_n$ and the final filename by `foo`. Since $\mathcal{M} \cap \mathcal{B} = \emptyset$ then set of writers/owners in those directories at the time where createOneName visited them was disjoint of $\mathcal{B}$. We next show that all these directories (and also the final file) remained unchanged since createOneName visited them, thus completing the proof.

First, we claim that at the time of creation, `fName` was a simple pathname. That `fName` does not include '`.`' or '`..`' or '`//`' follows since createOneName does not create names that include any of them. Also, uncorrupted processes in our implementation never create symbolic links, so symbolic links can only be created in directories that are writable by some role in $\mathcal{B}$. This means that none of the directories $\texttt{dir}_i$ contained symbolic links when the name-resolution visited them during createOneName, so in particular all the pathname components visited (or created) by createOneName (except the final `foo`) were directories. Once these directories were visited, they were not moved (since only corrupted processes can move directories but none of them had permission to do so), hence by Lemma 6 they also not removed before the hard link `foo` was created. Hence also at the time that `foo` was created, the pathname `fName` was simple.

Next, assume toward contradiction that one of the directories $\texttt{dir}_i$ (or `foo`) was modified or erased since it was visited by createOneName, and consider the first of them that was modified or erased. By Assumption 2, the caller owned or had write permission in the parent directory at the time of the change. Since the set of owners/manipulators is disjoint of $\mathcal{B}$, it means that the process that first modified/erased that pathname component must have been uncorrupted.

In our implementation, system calls that modify permissions are used by uncorrupted processes only on ephemeral names, which `fName` is not. Therefore the first modification had to be a removal of a pathname component (by an uncorrupted process). Invoking Lemma 6 again, we know that none of the directories can be removed, thus the first pathname element to be deleted has to be the hard link `foo` itself. Note also that hard links to files are only deleted by uncorrupted processes during a successful `DeleteName` call to SimpFS.

Denote the pathname argument to the successful `DeleteName` call that deleted `foo` by `fName2`, and we argue that `fName2` must be the same as `fName`. Clearly, `fName2` cannot include '`.`' or '`..`' or '`//`' since safeDirOpen does not allow these. Also, recall that the deleteName procedure was run by a process that had permissions to delete the hard link

foo, so it must have a different effective-uid from all the corrupted processes. Since only the adversary creates symlinks, then symlinks must reside in directories that are unsafe for the effective-uid of that process, hence safeDirOpen will not follow them. Therefore safeDirOpen encountered only hard links (to directories) as it resolved the name fName2.

We now argue that these directories must have been the same $\texttt{dir}_1,\texttt{dir}_2,\ldots,\texttt{dir}_n$ as in fName, and moreover at the time of deletion the hard-link must have been called foo (as in fName). For foo itself, we already established above that the first modification to it since it was created was the time it was removed. Hence at the time of deletion it must have been called foo and must have resided at deletion in the same directory in which is was created.

As for the containing directories, at the time that foo was created none of them was writable or owned by corrupted players, which implies that none of them was writable or owned by corrupted player any any point since these directories themselves were created. (This follows from Corollary 5.) Thus these directories could not have been moved to their containing directory at fName, they must have been created there with ephemeral names and then renamed to their permanent name, which remained fixed at least as long as foo existed. By induction on the pathname components of fName (starting from $\texttt{dir}_n$ and going back), we therefore conclude that the deleteName procedure must have opened each $\texttt{dir}_i$ using a handle to $\texttt{dir}_{i-1}$ and the same name that $\texttt{dir}_i$ has in fName. Hence fName2 and fName are the same.

Summing up, we had an uncorrupted player who made a successful call to the procedure DeleteName(fName) in the SimpFS functionality. But this means that fName no longer exists in SimpFS, which is a contradiction. □

Lemma 9 *At any point in time, two non-ephemeral protected names resolve to the same file in the simulated real world if and only if they belong to the same file in the* SimpFS *functionality.*

Proof: Fix any two non-ephemeral protected names that exist at some point in time in the simulated real world. By Lemma 7 they also exist in the SimpFS functionality. For each name, we look at the CreateFile call when it was *last* created in the SimpFS functionality, which was after the link system call returned successfully in the respective simulated createOneName subroutine.

If both createOneName subroutines were part of the same createFile procedure then they were created pointing to the same ephemeral filename, and since they are protected then also the ephemeral name was protected, which means that it was not deleted between the two link system calls. Hence they were created pointing to the same file. On the other hand, if the two subroutines were part of two different runs of createFile then they were created pointing to different files. By Lemma 8, the two pathnames remained unchanged since they were created. Hence, they still resolve to the same file if they were created in the same CreateFile call to the SimpFS functionality (and hence belong to the same file in SimpFS), and they still resolve to different files if they were created in two CreateFile calls (and hence belong to different files in SimpFS). □

Lemma 10 *Consider a call* `Write(fName,...)` *from an uncorrupted process that returns* `OKAY`*, and consider the state of the simulated real world at the time when the* `open` *system call in the implementation returns a handle to the final hard link. If at that time* `fName` *is unprotected, but there exists a protected name that resolves to the same file, then the file itself is unprotected (i.e., there is some role in* $\mathcal{B}$ *with permission to write in it).*

Proof: If any `root` process is corrupted then all files and names are unprotected and we are done. Assume from now on that no `root` process is corrupted. It follows that when the last `open` system call returned, the name `fName` was not system-safe (else it would have been protected), so the `Write` procedure did not open `fName` in a system-safe mode. Let U denote the effective-uid of the calling process. The same argument as above shows that if no U process was corrupted (when the `open` system-call returned), the `Write` procedure could not have opened `fName` in a safe-for-U mode. Hence the only two cases that we need to consider are that some U process was corrupted, or that `safe-open` opened `fName` in unsafe mode.

In the former case, recall that `fName` was not opened in system-safe mode, so a `Write` could only succeed when the file is either world-writable or owned by U (and writable by owner). Either way the file is not protected (since it can be written by the corrupted U process). It is left to show that the latter case (where the file was opened in unsafe mode and no U process is corrupted) cannot happen.

Since the file had a protected name it also had a *simple* protected name, which we denote `fName2` = `/dir1/.../dirn/foo`. The hard link `foo` must also be the last hard link in `fName`, as opening a file in unsafe mode would fail if the file has multiple hard links. Finally, the resolution of `fName` could not have encountered directories unsafe for U *before* merging into the simple path `fName2`, else it would fail. But since no U or `root` process is corrupted, all these directories were still safe for U when the `open` system call returned, hence `fName` was protected, which is a contradiction. □

Lemma 11 *The view of the environment is identical in the real and ideal worlds.*

Proof: We need to show that the answers that the environment sees when interacting with simpfs over POSIX and the adversary $\mathcal{A}$ are identical to what it sees from the SimpFS functionality with the simulator $\mathcal{S}$ and the same $\mathcal{A}$. Below we will argue about the simulated real world, since it is an exact replica of the real world.

From the description of the simulator, it is clear than whenever the implementation of some call returns an error code then the environment will see the same error code in the ideal world (since this is what the simulator returns to SimpFS). Also, it is clear that the results of all the calls that have *unprotected names* as arguments must be the same, since the simulator always creates the corresponding files in SimpFS on the fly to ensure this.

It is left to show that for operations that have *protected names* as argument, if they succeed in the real-world implementation then SimpFS will not return an error, and also that the content of successful `Read` operations is the same. We begin with error codes: The cases where the call to `AdversaryAction` returns `OKAY` but SimpFS returns an error are the following:

- In `CreateFile` when the filename already exists. By Lemma 8, if a protected filename exists in SimpFS then it also exists in the simulated real world, hence the `link` system call would fail and the simulator would not return `OKAY`.

- In `DeleteName/Read/Write` where the name does not exist, or the calling process does not have permission to delete the name or read/write the file.

  Recall that if a name does not exist but the operation in the real world succeeds, then the simulator creates the corresponding name with the right permissions in the SimpFS functionality before returning `OKAY`. So the only case that needs to be examined is when the name does exist (and does not have corrupted manipulators) but the calling process does not have the permissions to delete, read, or write. By Lemma 8, such names exist also in the simulated real world, and they remained unchanged since they were created. Moreover the createFile procedure ensures that the name and file have the same sets of manipulators/writers in the simulated real world as in the SimpFS functionality. Hence, if the calling process does not have permission to delete/read/write then the simulated procedure will also fail, and the simulator will not return `OKAY`.

Next we consider the content of files with protected names. By Lemma 8 this name also exists in the SimpFS functionality. We observe that the last time `fName` was created in the SimpFS functionality (prior to the successful `read` system call) could not have been between the `open` and `read` system calls, since otherwise the final `lstat` check would have failed and the `Read` would not have been successful. Hence the name (and the file) were created before the `open` system call.

We now examine the content of the file corresponding to `fName` since the last time it was created in the simulated real world. (This was when the temporary name for this file was created.) For each successful `write` system call for this file, we designate the beginning of the next successful `read` or `write` system call (for the same file) as "the point where the `write` operation ended." We prove by induction that at the time each `write` ended, the content of the file in SimpFS was identical to its content in the simulated real world.

We have two cases to consider: either the file is unprotected (i.e., one of the bad roles in $\mathcal{B}$ belongs to the `Writers` set), or it is protected. If the file is unprotected then the simulator would always make sure to adjust its content in the SimpFS functionality to whatever it would be in the simulated real world. We now claim that the last remaining case — where the file is protected but the name that was used to write in it is not — cannot happen.

If the `open` system call for the `Write` operation happened after the name `fName` was created in the simulated real world then we meet the conditions of Lemma 10, namely a successful `Write` to an unprotected name where the same file also has a protected name (the protected name is `fName`). If the `open` system call happened before the name `fName` was created then the temporary name for that file must have still existed at the time, which was itself protected, and again we meet the conditions of Lemma 10. In either case the file cannot be protected.

We have shown that the content of the file is identical at the end of every `write` operation. Since the `open` call for the `Read` happened after the file was created then the subsequent `read`

system call returns the content of this file (specifically, the content after the last `write` system call), which is the same as the content that SimpFS has for that file. This completes the proof of Lemma 11 and also Theorem 1. □

5.6 Summary

In this work we adapted the Universal Composability (UC) framework to the modeling of large software systems. Focusing on filesystem interfaces, we described SimpFS, which is a simple filesystem abstraction intended to capture *filesystem integrity* concerns. We describe an implementation of this abstraction over real POSIX filesystems and prove that the implementation realizes the SimpFS abstraction in the UC sense. SimpFS is a simple but useful interface and with a few small enhancements is sufficient to build real applications.

Our work demonstrates that formal security frameworks such as Universal Composability can also be used beyond the niche of cryptographic protocols. Our modeling of POSIX-based file systems is the first example of this scale. Our proof implies that it is possible for applications to enjoy the security assurances of an idealized system interface even when running over a large complex interface (and even though potential attackers can use the entire larger interface). Moreover, the composability gurantees of UC allow us to retain this assurance irrespective of what the application is.

6 Implementation of a safe Filesystem primitive and its analysis

In the previous section we used the UC framework to identify a simple file system interface which was an idealization of a safe way to use the POSIX filesystem interface. In this section we consider practical realizations of this idea and aim to define practical file system primitives using the ideas from the previous section. We will further evaluate our implementation on real systems to evaluate the practicality of our solution.

In this work we take a closer look at the problem of privilege escalation via manipulation of filesystem names. Historically, attention has focused on attacks against privileged processes that open files in directories that are writable by an attacker. One classical example is email delivery in the UNIX environment (e.g., [Mai]). Here, the mail-delivery directory (e.g., `/var/mail`) is often group or world writable. An adversarial user may use its write permission to create a hard link or symlink at `/var/mail/root` that resolves to `/etc/passwd`. A simple-minded mail-delivery program that appends mail to the file `/var/mail/root` can have disastrous implications for system security. Other historical examples involve privileged programs that manipulate files under the world-writable `/tmp` directory [ope02], or even in a directory of the attacker's choice [xte93].

Over time, privileged programs have implemented safety mechanisms to prevent path-name resolution attacks. These mechanisms, however, are tailored specifically to the program's purpose, are typically implemented in the program itself, and rely on application-specific knowledge about the directories where files reside. We believe, however, that the application is fundamentally the wrong place to implement these safety mechanisms.

Recent vulnerability statistics support our position. The US National Vulnerability Database [nvd] lists at least 177 entries, since the start of 2008, for symlink-related vulnerabilities that allow an attacker to either create or delete files, or to modify the content or permissions of files. No doubt, the vast majority of these entries are due to application writers who simply were not aware of the problem. However, there are even vulnerabilities in system programs, which are typically better scrutinized. For example, an unsafe file `open` vulnerability was reported in the `inetd` daemon in Solaris 10 [sol08] when debug logging is enabled. This daemon runs with `root` privileges and logs debug messages to the file `/var/tmp/inetd.log` if that file exists. The file is opened using `fopen(DEBUG_LOG_FILE, "r+")`. Since `/var/tmp` is a world writable directory a local unprivileged user can create a link to any file on the system, and overwrite that file as `root` with `inetd` debug messages. A similar example, related to unsafe `unlink` operation, is a reported vulnerability in the Linux `rc.sysinit` script [rcs08] in the `initscripts` package before version 8.76.3-1. That vulnerability could be used by unprivileged users to delete arbitrary files by creating symbolic links from specific user-writable directories.

In addition to these examples, experiments that we run in the course of this work uncovered a number of (latent) privilege escalation vulnerabilities, where system processes write or create files as `root` in directories that are writable by unprivileged system process. In these cases, a compromise of the unprivileged system process could result in further privilege

escalation. These vulnerabilities are described in Section 6.5.3.

These examples demonstrate that it is unrealistic to expect every application (or even every "important application") to implement defenses against these attacks. We contend that a system-level safety net would be more effective at stopping these problems than trying to fix every affected application, or trying to educate current and future generations of application writers. In a world where applications (and their fragments) are used in environments that are vastly different from what the application designers had in mind, it is unreasonable to expect that the applications themselves will distinguish between files that are safe to open and ones that are not.

In this work we seek a general-purpose mechanism that can be implemented in the file system or in a system library, that allows programs to open files that exist in an "unsafe" environment, knowing that they will not be "tricked" into opening files that exist in a "safe" environment. Specifically, we show how such a mechanism can be implemented over POSIX filesystems.

In a nutshell, our solution can be viewed as identifying "unsafe subtrees" of the filesystem directory tree, and taking extra precautions whenever we visit any of them during the resolution of a pathname. Roughly, a directory is unsafe for a certain user if anyone other than that user (or `root`) can write in it. Our basic solution consists of resolving a pathname component by component, enforcing the conditions that once we visit an unsafe node, in the remainder of the path we will no longer follow symbolic links or allow pathname elements of '..', nor will we open a file that has multiple hardlinks. Thus, once we resolve through an unsafe node, we will not visit nodes that exist outside the subtree rooted at that node.We describe in Section 6.6.1 a more permissive variant that still provides the same protection against privilege-escalation attacks.

In contrast with many prior works on filename-based attacks, our work is *not primarily focused on race conditions* (such as access/open races [THWS08b, CGJ09]). Rather, we directly addresses the privilege-escalation threat, which is the main motivation for many of these attacks. Here we focus on the pathname resolution mechanism, identify a simple security property that can be met *even in the presence of race conditions*, and show that this property can be used to prevent privilege-escalation attacks.

We focus on tightening the connection between files and their names. In most filesystems, programs access files by providing names (the pathnames), and rely on the filesystem to resolve these names into pointers to the actual files (the file handles). Unfortunately, the relation between files and their names in POSIX filesystems is murky: Files can have more than one name (e.g., due to hard or symbolic links), these names can be changed dynamically (e.g., by renaming a directory), filename resolution may depend on the current context (e.g., the current working directory), etc. This murky relation obscures the semantics of the name-to-file translation, and provides system administrators and applications writers with ample opportunities to introduce security vulnerabilities. Our solution builds on the following concepts:

- Ignoring the partition to directories and subdirectories, we view the entire path as just one name and examine its properties. We introduce the concept of *the manipulators* of

a name, which roughly captures "anyone who can change the outcome of resolving that name." In POSIX filesystems, the manipulators of a path are roughly the users and groups that have write permission in any directory along this path. More precisely, U belongs to the manipulators of a name if the resolution of that name visits any directory that is either owned by U or that U has write permissions for.

- Using the concept of manipulators, we distinguish between *safe names* and *unsafe names*. Roughly, a name is safe for some user if only that user can manipulate it. Specializing this concept to UNIX systems, we call a name "system safe" if its only manipulator is `root`, and call it "safe for U" if the only manipulators of it are `root` and U. For example, typically the name `/etc/passwd` is "system safe", the name `/home/joe/mbox` is safe for user `joe`, and the name `/var/mail/jane` is not safe for anyone (as `/var/mail` is group or world writable).

- Once we have safe and unsafe pathnames, we can state our main security guarantee. We provide a procedure `safe-open` that ensures the following property:

  > *If a file has safe names for user U, then* `safe-open` *will not open it for U using an unsafe name.*

  As we show in the section, this property can be used to ensure that no privilege escalation via filesystem links occurs. For example, if `/etc/passwd` is system-safe, then no process running as root will `safe-open` this file due to a hard link or symbolic link that could have been created by a non-`root` process. In particular, a "simple minded" mail delivery program that uses our `safe-open` will be protected against the attack in the example from above. Also, we verified that this guarantee is sufficient to protect against the documented vulnerabilities in CVE.

We implemented our `safe-open` procedure as a library function over POSIX file systems, and also generalized it to other POSIX interfaces that resolve pathnames such as `safe-unlink`, `safe-chmod`, etc. (cf. Section 6.4). We performed whole-system measurements with several UNIX flavors, and find that system-wide safe pathname resolution can be used without "breaking" real software. During these measurements we also uncovered a number of new (latent) vulnerabilities (cf. Section 6.5.3), that would be fixed using our `safe-open`.

6.1 Related Work

Much of the prior work on pathname safety has focused on time-of-check/time-of-use race vulnerabilities (TOCTTOU) in privileged programs [Bis95, BD96, DH04, BJSW05, THWS08b, CGJ09]. Our work is not focused on this problem, instead it directly addresses the privilege-escalation issue that underlies many of these race-condition vulnerabilities: Rather than trying to prevent race conditions, we modify the name-resolution procedure to ensure that privilege-escalation cannot happen even if an attacker is able to induce race conditions.

In early analysis of filesystem race vulnerabilities in privileged programs, Bishop discusses safe and unsafe pathnames, and introduces a `can-trust` library function that determines whether an untrusted user could change the name-to-object binding for a given pathname [Bis95]. Later, a more formal analysis with experimental validation was done by Bishop and Dilger [BD96].

Our `safe-open` function implements a user-level pathname resolver that examines pathname elements one by one; its structure is therefore similar to that of the `access-open` function by Tsafrir *et al.* [THWS08b, THWS08a]. While their user-level name resolver applies access checks to each path element in a manner that defeats race attacks, our `safe-open` function is not primarily concerned with access checks. Instead, we apply a "path safety" check to each path element.Our solution could have been implemented using a variant of the general framework from [THWS08a, Sec. 7], but that variant would have to be considerably more complex to deal with issues such as change of privileges or permissions, thread safety, etc.

In the context of system call introspection monitors for TOCTTOU vulnerabilities, Garfinkel [Gar03] considered remedies which could also potentially apply to the problem of unsafe pathname resolution. These remedies include disallowing the creation of symlinks to files which the calling process does not have write permissions to, as well as denying access to files through symlinks. As noted in his paper, these solutions can mitigate the problem but they do not solve it. For instance, they do not address pre-existing symlinks, and fail in the face of symlinks in intermediate components of the pathnames. In contrast, our solution directly addresses the underlying problem of unsafe pathname resolution.

Another approach to system call introspection is implemented in the Plash sandboxing system [Sea]. Here, a replacement C library delegates file-system operations to a fixed-privilege, user-level, process that opens files on behalf of monitored applications and that enforces a confinement policy. While this approach provides great expressiveness, it would not be suitable for system-wide deployment as envisaged with our `safe-open` function. (For example it is not clear how to address privilege changes by the calling process, or how this solution scales with the number of processes.)

Addressing filename manipulations is in some ways complementary to dealing with the "confused deputy" problem: Both problems are used as a vehicle for privilege escalation, and some aspects of the solution are common, but the problems themselves appear to be different: For example, the "simple minded" mail-delivery program from above knows that it uses its `root` privileges for writing `/var/mail/root`, so in this sense it is not a confused deputy (since it is not being tricked into using some extra privilege that it happens to hold). The problems with UNIX privilege-managing functions were systematically analyzed by Chen, Wagner and Dean; these authors also provide a more rational API for privilege management [CWD02]. Their approach was later extended by Tsafrir, Da Silva and Wagner to include also group privileges [TSW08].

Mazieres and Kaashoek advocate a better system call API that among others allows processes to specify the credentials with each system call [MK97]. Our `safe-open` function could benefit from such features (especially when opening files on behalf of `setgid` programs,

cf. Section 6.6.3).

6.2 Names, Manipulators, and Safe-Open

For presentation simplicity, we initially consider only a simplified setting where (a) all filenames are absolute paths, (b) every filesystem is mounted only once in the global name tree, and (c) no concurrency issues are present. (The last item means that we simply assume that no permission changes occur concurrently with our name resolution procedure.) We discuss relative pathnames at the end of this section, multiple mount points and dynamic permissions are discussed in Section 6.3.

6.2.1 Names and Their Manipulators

Roughly speaking, a *manipulator* of a name is any entity that has filesystem permissions that can be used to influence the resolution of that name. A manipulator can create a name (i.e., cause the filesystem to resolve that name to some file), delete it (causing name resolution to fail) or modify it (causing the name to be resolved to a different file). In the context of POSIX systems, a *manipulator* of a path in a POSIX filesystem is any `uid` that has write permission in — or ownership of — any directory that is visited during resolution of that path.See Section 6.6.3 for a discussion about `gid`s.

For example, consider the files `/etc/passwd`, `/home/joe/mbox`, and `/tmp/amanda/foo` from a common UNIX system. The permissions of the relevant directories are:

```
drwxr-xr-x root root /
drwxr-xr-x root root /etc
drwxr-xr-x root root /home
drwx------ joe  joe  /home/joe
drwxrwxrwt root root /tmp
drwxr-xr-x root root /tmp/amanda
```

Then the only manipulator of the name `/etc/passwd` is `root` (since only `root` can write in either `/` or `/etc/`), and the manipulators of the name `/home/joe/mbox` are `root` and `joe`. On the other hand, all the users on that machine are manipulators of `/tmp/amanda/foo`, since everyone can write in `/tmp`.The directory `/tmp` typically has the sticky bit set, which prevents non-`root` users from removing other user's files from `/tmp`. But it does not prevent users from moving other user's files *into* `/tmp`. For this reason, everyone must be considered a manipulator of the directory `/tmp/amanda`, even though this directory can be removed only by `root`.

Moreover, if we had the symbolic links: then the manipulators of `/home/joe/link1` are `root` and `joe`, and the manipulators of the name `/home/joe/link2/foo` include all the users on that machine (since resolution of this last name goes through the world-writable `/tmp`).

We note that this description is "static", in that it refers to the permission structure as it exists at a given point in time. Nonetheless, in Section 6.3.2 we show that our solution (which

is based on this "static" notion) prevents privilege escalation via pathname manipulations even in settings where the filesystem (and its permissions) can change in a dynamic fashion. Roughly speaking, this is because in POSIX systems only manipulators of a path can add new manipulators to it, and no manipulator can remove itself from the set of manipulators of a path.The last statement depends on the fact that only `root` can use the `chown` system call.

Safe and unsafe names. For POSIX systems, we say that a name is *system-safe* (or safe for `root`) if `root` is the only manipulator of that name. A name is safe for some other `uid` if its only manipulators are `root` and `uid`. Otherwise the name is *unsafe*.

6.2.2 The `Safe-Open` Procedure

Our `safe-open` procedure is a refinement of the safety mechanisms used by the Postfix mail system [Ven] to open files under the world-writable directory `/var/mail`. The basic approach taken by Postfix is to verify that the opened file is not a symbolic link and does not have multiple hard links. This approach works for the special case of `/var/mail`, but it is not quite applicable as a general-purpose policy, for two reasons:

It is too strict. There are cases where applications have a legitimate need to open a file with multiple hard links or a symbolic link.For example, old implementations of Usenet news kept a different directory for every newsgroup and a different file for every article, and when an article was sent to more than one group, then it will be stored with multiple hard links, one from each group where this article appears. Moreover, blanket refusal to open files with multiple hard links would enable an easy denial-of-service attack: simply create a hard link to a file, and no one will be able to open it.

It is not strict enough. Refusing to open links does not provide protection against manipulation of higher-up directories. For example, consider a program that tries to open the file `/tmp/amanda/foo`. Even if this file does not have multiple links, it may still not be safe to open it: For example, the attacker could have created `/tmp/amanda/` as a symbolic link to `/etc`, and the program opening `/tmp/amanda/foo` will be opening `/etc/foo` instead.

To implement a general-purpose `safe-open`, we therefore refine these rules. Our basic procedure is as follows: While resolving the name, we keep track of whether the path so far is *safe* or *unsafe* for the effective `uid` of the calling process. When visiting a directory during name resolution, we call it unsafe if it is group- or world-writable, or if its owner is someone other than `root` or the current effective `uid` of the calling process (and otherwise we call it safe). When resolving an absolute path, we start at the root in safe mode (if the root directory is safe). As long as the resolver only visits safe directories, we are in a safe mode, can follow symbolic links or '`..`', and can open files with multiple hard links. However, once the resolver visits an unsafe directory, we switch to unsafe mode, and in the remainder of the

path, disallow symbolic links or '..', and refuse to open a file with multiple hard links.See Section 6.6.1 for more permissive variants of this procedure. We note the following about this solution:

- A safe name that can be opened by POSIX `open` will also be opened by `safe-open`: If a name is safe then the `safe-open` procedure will visit only safe directories, and therefore will not abort due to symlinks or multiple hardlinks. Any directory that is visited during name resolution in `open` will also be visited by `safe-open`, and the file will eventually be opened.

- A file with only one name (which can be opened by POSIX `open`) will be opened by `safe-open`: This is similar to the previous argument, if the file has just one name then this name cannot include symbolic links and the file cannot have multiple hard links. Hence `safe-open` will succeed in opening it if POSIX `open` does.

- For a file with multiple unsafe names, each of these names may or may not be opened by `safe-open`. Note that if many names point to the same file, then there must be "merge points" where either we have a symbolic link pointing to a directory (or to the file) or multiple hard links pointing to this file. When `safe-open` resolves these names, it agrees to follow these "merge points" if it visited only safe directories before they occur, and refuses to follow them if it visited an unsafe directory.

  For one example, `safe-open` will agree to open the unsafe name `/home/joe/link2/foo` from Section 6.2.1 when running with effective `uid` of `joe`, since the "merge point" occurred while visiting the directory `/home/joe/`, still in a safe mode. On the other hand, `safe-open` will refuse to open this name when running with effective `uid` of `root`, since the directory `/home/joe/` is not safe for `root`.

Implementing this `safe-open` procedure in the filesystem itself (i.e., in the kernel) should be straightforward: All we need is to add a check for permissions and ownership on every directory, updating the safety flag accordingly. Arguably, this is the preferred mode of implementation, but it requires changes to existing filesystems. Alternatively, we describe an implementation of `safe-open` as a library function in user space. This implementation roughly follows the procedure of Tsafrir et al. [THWS08b, THWS08a] for user-level name resolution, but adds to it the safe-mode vs. unsafe-mode behavior as described above. We discuss this implementation in Section 6.4.

Relative paths and `openat`. The procedure for resolving relative paths (or for implementing `openat`) is essentially the same as the one for absolute paths, except that we need to know if the starting point (e.g., the current working directory) is safe or not. In a kernel implementation, it is straightforward to keep track of this information by adding flags to the handle structure. Some care must be taken in situations where the directory permissions change (e.g., via `chmod` or `chown`) or when the privileges of the current process change, but no major problems arise there. Keeping track of this information in a library implementation

is harder, but even there it is usually possible to get this information, and reasonable defaults can be used when the information is unavailable (e.g., after an `exec` call). Section 6.7 provides more details about relative paths and `openat`.

6.3 Our Security Guarantee

Recall the security guarantee that we set out to achieve:

If a file has names safe for user U, then `safe-open` *will not open it for U using an unsafe name.*

In other words, if a file has both safe and unsafe names, then `safe-open` should fail on all the unsafe names. (At the same time it succeeds on all the safe names, as noted above.) We note again that as stated, this guarantee applies only to a static-permission model, where permissions and ownership of directories do not change during the name resolution. However, as we discuss at the end of this section, protection against privilege escalation attack is ensured *even when the attacker makes arbitrary permission changes for directories that it owns.* The only thing that we must assume is that non-adversarial entities do not induce a permission-change race against our name resolution.The distinction between adversarial and non-adversarial entities is inherent in privilege-escalation attacks, since one must distinguish between privileges held by the attacker and those held by the victim(s).

Our analysis below also assumes that each directory tree appears only once in the file system tree (i.e. no loop-back mounts, etc.), and that each directory has at most one parent (i.e., one hard link with a name other than '`.`' or '`..`').Nearly all contemporary POSIX implementations either do not allow processes to create additional hard links to directories (e.g., FreeBSD, Linux) or restrict this operation to the super-user (e.g., Solaris, HP-UX). A notable exception is MacOS. A short discussion of mount points can be found later in this section.

We now turn to proving this security guarantee. Consider a file that has both safe and unsafe names (for a specific `uid`), fix one specific unsafe name, and we show that `safe-open` must fail when it tries to open that name (on behalf of a process with this effective `uid`). We distinguish two cases: either the file has just one hard link, or it has more than one.

- *Case 1: more than one hard link.* Note that when `safe-open` is called with the unsafe name, it will apply name resolution while checking the safety of the name as it resolves it. As the resolution of this name goes through a directory which is unsafe for this `uid`, then `safe-open` will arrive at the last directory in this name resolution in unsafe mode (assuming that it arrives there at all). Since the file has more than one hard link, `safe-open` will then refuse to open it.

- *Case 2: exactly one hard link.* In this case, there is a single path from the root to this file in the directory tree (i.e. we exclude names that contain symbolic links). Below we call this the "canonical path" for this file and denote it by `/dir1/dir2/.../dirn/foo`.

Clearly, every pathname that resolves to this file must visit all the directories on the canonical path. (Moreover, the last directory visited in every name resolution must be `dirn`, since it holds the only hard link to `foo`.) Since we assume that the file has safe names for `uid`, it follows that all the directories in this canonical path must be safe for `uid`.

Consider now the directories visited while resolving the unsafe name. Being unsafe, we know that the resolution of this name must visit some unsafe directory, and that unsafe directory cannot be on the canonical path. Therefore, during the resolution of an unsafe name, `safe-open` must visit some unsafe directory (and therefore switch to unsafe mode) before arriving at the final directory `dirn`.

Consider the last directory *not on the canonical path* that was visited while resolving this unsafe name. We call this directory `dir0`. Then it must be the case that `safe-open` switched to unsafe mode when visiting `dir0` or earlier (because after `dir0` it only visited safe directories). Now, since the canonical path begins with the root '`/`', then `safe-open` could not descend into the canonical path from above. Hence moving from `dir0` to the next directory was done either by following a symbolic link or by following '`..`', but this is impossible since `safe-open` does not follow symbolic links or '`..`' when in unsafe mode.

This completes the proof of our security guarantee.

Multiple mount points. We note that all the arguments from above continue to hold even when a filesystem is mounted at multiple points in the global name space, as long as all the mount points are system-safe. However, our security guarantee breaks if we have the same filesystem mounted in several directories, some safe and others not. In this case, going down a "canonical" unsafe name for a file, we have no way of knowing that the same file also have a safe name (via a different mount point). The same problem arises when parts of the filesystem are exposed to the outside world, e.g., via NFS. In this case, what may appear as a safe directory to a remote user may be unsafe locally (or the other way around).

6.3.1 Using the Security Guarantee to Thwart Privilege Escalation

The security guarantee that we proved above provides one with an easy way of creating files that applications cannot be "tricked" into opening using adversarial links: Namely, create the file with a safe name. For example, if the name `/path-to/foo` is system safe, then no process running as `root` can use `safe-open` to open the same file with a name that includes a link that was created (or renamed, or moved to its current location) by a non-`root` user. This is because such a link would have to be created in (or moved to) an unsafe directory, making the name unsafe and causing `safe-open` (running as `root`) to fail on it.

This observation can be used to defeat privilege escalation attacks. Consider a file that needs to be protected against unauthorized access (where access can be read, write, or both). Hence the file is created with restricted access permissions. To ensure that this protection

cannot be overcome by the attacker creating adversarial links, we create this file with a name that is safe for all the `uid`s that have access permission for it. (That is, if only one `uid` has access permission to the file then the name should be safe for that `uid`, and otherwise the name should be system-safe.)

We now claim that an attacker that cannot access the file, also cannot create a link that would be followed with `safe-open` by anyone with access permission for this file. Note that the attacker must have a different `uid` than anyone who can access the file.See Section 6.6.3 for a short discussion of `setgid` programs.

Hence a directory where the attacker can create a link must be unsafe for anyone who can access the file, and therefore `safe-open` will not follow links off that directory.

6.3.2 Dynamic Permissions

The argument above covers the static-permission case, where permissions for directories do not change during the execution of `safe-open`. We now explain how it can be extended to the more realistic dynamic-permission model.

Consider a potential privilege-escalation attack, where an attacker that cannot access a certain file tries to cause a victim program to access that file on its behalf. Notice that in this scenario it must be the case that the attacker does not have `root` privileges, and also has a different effective-`uid` than the victim. (Otherwise no privilege escalation is needed — the attacker could access the file by itself.

Consider now a file F that can be accessed by the effective-`uid` of the victim (denoted by U) but not by the effective-`uid` of the attacker (denoted by U'), consider a particular execution of `safe-open` by the victim, and assume that:

(a) at the time that the procedure is invoked, the file F has some name that is safe for U, and that name remains unchanged throughout the execution, and

(b) the pathname argument to `safe-open` is not a U-safe name for the file F when the procedure is invoked.

Under these conditions, we show that this `safe-open` procedure will not open the file F, *barring a concurrent filesystem operation by `root` or U on pathname elements that `safe-open` examines.* Put in other words, the attacker can only violate our security guarantee if it can induce a race condition *between two non-adversarial processes* (i.e., the `safe-open` procedure and another process with `uid` of either the victim or `root`). Assume therefore that these two conditions hold, and in addition

(c) neither `root` nor U did any concurrent filesystem operation on any pathname element examined by this `safe-open`.

We observe that any pathname element that `safe-open` examines and that resides in a U-safe directory at the time where the procedure was invoked, must remain in the same state throughout this `safe-open` execution. The reason is that being U-safe, only U and `root`

have permissions to change anything in the directory, and by our assumption (c) neither of them made any changes to that pathname element.

Imagine now that the state of the filesystem is frozen at the time when the `safe-open` procedure is invoked, and consider the way the pathname argument to `safe-open` would be resolved. We have two cases: either all the directories visited by this hypothetical name resolution are U-safe, or some of them are not. The easy case is when all of them are U-safe: then it must be the case that the hypothetical name resolution does not resolve to the file F (or else it would be a U-safe name for F, contradicting our assumption (b)). But it is easy to show (by induction) that the same directories will be visited also in the actual name resolution, all of them would be in exactly the same state, and therefore also the actual name resolution as done by `safe-open` would not be resolved to F.

Assume, then, that the hypothetical name resolution would visit some unsafe directories, and let `dir0` be the first U-unsafe directory to be visited. The same easy inductive argument as above shows that all the directories upto (and including) `dir0` are also visited by the actual name resolution. We now know that the owner of `dir0` remains the same throughout the execution of `safe-open` (since by assumption (c) `root` did not make any changes in directories that were examined by `safe-open`). If the owner is different than U and `root`, then `safe-open` will switch to unsafe mode when it gets to `dir0`. If the owner is U or `root` then it must be the case that the directory was group- or world-writable when `safe-open` was invoked (since it was unsafe in the hypothetical resolution), and thus it must still be group- or world-writable when `safe-open` examines it (since by our assumption (c) U and `root` did not change that directory). We therefore conclude that the hypothetical and actual name resolutions proceeded identically upto (and including) `dir0`, and they both switched to unsafe mode upon visiting `dir0`.

In particular it implies that `safe-open` arrived at the final directory in unsafe mode, so it would only open F if F had a single hard link at the time that the procedure returned. Recall now that by our assumptions (a), this single hard link must be at the end of a U-safe pathname. But we know that `safe-open` visited at least one unsafe directory, so its traversal must have merged back into the safe pathname at some point after visiting `dir0`. As in the static case, this must have happened by following a symbolic link or '`..`', which is a contradiction.

Preventing privilege-escalation in the dynamic setting. Once we established the security guarantee in the dynamic setting, we can show how to use it to prevent privilege escalation even in a filesystem where permissions can change. In addition to creating the protected files with safe names, we also need to ensure that (a) we never reduce the write permissions of a non-empty directory that was group- or world-writable or `chown` a non-empty user directory back to `root`; and (b) we do not change permissions or ownership in the safe name and do not delete it while there are still programs that have the file open.

It is not hard to see that as long as (a) and (b) do not happen, then the conditions that we set in our dynamic-system proof hold, and hence no privilege-escalation can result from adversarial filesystem actions. Seeing that condition (a) is really needed is also easy:

indeed if the attacker creates an adversarial link in a world-writable directory and then the victim `chmod`s the directory and removes the world-writable permission, then `safe-open` will happily follow the adversarial link. Demonstrating that (b) is needed is a bit more tricky: Consider for example the file `/etc/passwd`, which is only writable by `root`, and consider the following sequence of operations:

1. Some user program P opens `/etc/passwd` for read and keeps the handle,
2. The attacker creates another hard link `/var/mail/root` to he same file,
3. A confused administrator deletes `/etc/passwd`, and
4. The mail-delivery program uses `safe-open` to open `/var/mail/root`, and then writes into it.

Note that `safe-open` will succeed under these conditions, since now `/var/mail/root` is the only name for this file (and in particular the file has only one hard link). But when the program P goes to read from its file descriptor, it will see the data that the mail-delivery program wrote there.

6.4 Implementing `safe-open` for POSIX Filesystems

We implemented `safe-open` as a library routine over the POSIX filesystem interface. The routine performs user-level name resolution, in a similar fashion as the routines of Tsafrir *et. al* [THWS08b, THWS08a], while adding the pathname safety check in every directory. That is, the routine goes through each component of the path to be opened, checks for the manipulators of each directory, and marks a directory unsafe if it has manipulators other than `root` and the current process' effective `uid`. Once it encounters an unsafe directory, in the remainder of the path, it does not follow symlinks or '`..`', and does not open a file with multiple hardlinks. A pseudocode description of our implementation is found in Figures 5 and 6.

6.4.1 Race conditions

Our name-resolution procedure is not particularly vulnerable to filesystem-based adversarial race conditions, in that it would correctly label safe/unsafe directories regardless of concurrent actions of any attacker (as long as the `euid` of the attacker is neither `root` nor the victim's `euid`). There are only two points in our code where we need to guard against check/use conditions:

(A) We must never `open` a symbolic link. If the `O_NOFOLLOW` flag is available then we can use it for that purpose, but to get get the same effect in a truly portable code we implement the `lstat-open-fstat-lstat` pattern.

(B) The other check/use window in our code is between the time that we check permissions and conclude that we are in a safe directory and the time that we read a symbolic

```
/* Resolve a pathname and open the target file */

safe_open(path, open_flags, is_safe_wd)
{
    if (path is absolute) {
        is_safe_wd = 1; dirhandle = null;
    } else {
        dirhandle = open(".", O_RDONLY) or return error;
    }
    return safe_lookup(dirhandle, path, is_safe_wd,
                       lookup_flags_for_open,
                       open_action_func, open_flags);
}

/* Call-back to open the final pathname component */

open_action_func(dirhandle, name, is_safe_wd, open_flags)
{
    truncate = (open_flags & O_TRUNC);
    flags = (open_flags & ~O_TRUNC);

    filehandle = openat(dirhandle, name, flags)
                 or return error;
    fst = fstat(filehandle) or return error;
    /* lstatat(args) is local alias for
       fstatat(args, AT_SYMLNK_NOFOLLOW) */
    lst = lstatat(dirhandle, name) or return error;

    if (fst and lst don't match) return EACCESS;
    check dirhandle permissions again,
          and update is_safe_wd if unsafe;
    if (!is_safe_wd && name is "..") return EACCES;
    if (!is_safe_wd && fst is not a directory
        && fst has multiple hard links)
        return EACCES;
    if (truncate) ftruncate(filehandle,0)
                  or return error;
    return filehandle;
}
```

Figure 5: The top-level `safe_open` and a utility function `open_action_func`.

link or open a file or directory. As we explained in Section 6.3.2, this check/use window is only open to races against processes with the same effective `uid` as the process calling `safe-open` (or `root`), not to races against an adversarial process trying to escalate privileges. As permission-changing actions by benign processes are quite rare, we believe that this window does not pose a major threat. We can even check the directory permissions both before and after reading a symlink (or opening a file or directory) to further narrow this window (and then this race cannot happen as long as non-adversarial processes do not revoke write permissions on non-empty directories).

6.4.2 Thread safety

Implementing user-level name resolution requires that we work with handles to directories, using either the current working directory (which may not be thread safe) or the `openat`,

```
/* Resolve pathname, invoke action on final component */

safe_lookup(dirhandle, path, is_safe_wd, lookup_flags,
            action_func, action_args)
{
    if (path is empty) return ENOENT;
    if (path is absolute) {
        dirhandle = open("/", O_RDONLY) or return error;
        lst = result of lstat("/") or return error;
        if (lst.owner not in [root, euid] || anyone not in [root, euid] can write)
            is_safe_wd = false;
        skip leading "/" in path, and replace path by "." if the result is empty;
    }
    while (true) {
        split path into first and suffix, and replace all-slashes suffix by ".";
        lst = result of lstatat(dirhandle, first) or return error;

        /* the meaning of "final pathname component" depends on lookup_flags, it has different *
         * meaning for open, unlink, etc */
        if (first component is final pathname component)
            return action_func(dirhandle, first, is_safe_wd, action_args);

        if (first component is a symlink) {
            newpath = readlinkat(dirhandle, first) or return error;
            check dirhandle permissions again, and return EACCES if unsafe;

            /* symlink at end of pathname */
            if (suffix == null)
               return safe_lookup(dirhandle, newpath, is_safe_wd, lookup_flags, action_func, action_args);

            /* other symlink */
            [newhandle, fst] = safe_lookup(dirhandle, newpath, is_safe_wd, lookup_flags, null, null)
              or return error;

        } else {
            /* first component is not a symlink */
            newhandle = openat(dirhandle,first,O_RDONLY) or return error;
            check dirhandle permissions again, and update is_safe_wd if unsafe;

            if (!is_safe_wd && name is "..")
               return EACCES;
            fst = result of fstat(newhandle)
                  or return error;

            if (first component is not a directory)
               return ENOTDIR;

            lst = result of lstatat(dirhandle, first)
                  or return error;
            if (lst does not match fst) return EACCES;

            /* reached the end of readlinkat result */
            if (suffix == null) return [newhandle, fst];
        }
        path = suffix;
        dirhandle = newhandle;
        if (fst.owner not in [root, euid] || anyone not in [root, euid] can write)
            is_safe_wd = false;
    }
}
```

Figure 6: The `safe_lookup` recursive call.

`readlinkat` and `fstatat` interfaces, which are part of a recent POSIX standard [ope08]. These interfaces duplicate existing pathname-based interfaces but add another parameter, a file descriptor for a directory. When used with a relative name, these calls now work relative to the specified directory instead of the current working directory.

The new interfaces are implemented in current Solaris and Linux versions. On systems without support for the `openat` family of function calls, we emulate their functionality inside a synchronized block: Maintaining a handle to the directory currently visited, we store the current working directory, change directory with `fchdir` to the visited directory, explore the next path element (for example, with `open` or `lstat`), then restore the original current working directory. To make the emulation signal-safe we also need to suspend signal delivery while in the protected block.

6.4.3 Read permissions on directories

Our user-level `safe-open` implementation relies on the ability to open all the intermediate directories (e.g., to `fstat` them or to use them with `openat`). Each path component, except the final one, is opened in a `O_RDONLY` mode. For this implementation to work, the process must have read permission on each non-final component in the path (in addition to the search permission that is required to look up the next pathname component in that directory). This is different from the regular POSIX `open` that only requires search permission on each directory component.

This restriction is of only temporary nature: a recent POSIX standard [ope08] introduces the `O_SEARCH` flag to open a directory for search operations only, and a future `safe-open` implementation can migrate to this.

6.4.4 Opening files without side effects

Upon arriving at the last path element (i.e., the file to be opened), our `safe-open` implementation may still need to verify that it is not a symbolic link. We again use the `lstat-open-fstat-lstat` pattern, but we must guard against potential side-effects of opening the file. For instance, opening the file with the flag `O_TRUNC` in combination with either `O_WRONLY` or `O_RDWR` will truncate the file before the `safe-open` procedure can determine that it opened an unexpected file. To fix this problem, we must first remove the `O_TRUNC` flag when opening the file, and if no error occurs then call `ftruncate` on the newly opened handle before returning it.

Somewhat similarly, if `safe-open` unexpectedly opens a target which is not a regular file (such as a `FIFO` or a `tty` port), then the open call could block indefinitely. This can be addressed only with cooperation by the application: when an application never intends to open a blocking target then it could specify the flag `O_NONBLOCK`.

6.4.5 Implementing `safe-create`, `safe-unlink`, and other primitives

Building on the same ideas, we can implement safe versions of other POSIX interfaces, such as `safe-create` for creation of new files, `safe-unlink` for removing them, etc. For many of these primitives, the implementation can be almost trivial: follow the same steps as with `safe-open` to reach the final directory; in the final step, `safe-create` creates the file (with flags `O_CREAT` and `O_EXCL`), and `safe-unlink` removes the target which may be a symlink or a file with multiple hardlinks.Some primitives (such as `unlink` and `mkdir`) do not follow a symlink that appears as the final pathname component; the `safe-unlink` and `safe-mkdir` functions must of course behave accordingly.

Our generalized pathname safety policy is easy enough to express: "*when resolving a pathname through an unsafe directory, in the remainder of the path don't follow '..' or symbolic links, and don't open or change attributes of files with multiple hardlinks.*" Articulating the exact security properties that you get may take some care. For example, the security property that you get from `safe-create` is this: "When called with an unsafe name, `safe-create` will fail to create the file if the resulting file could also have a safe name."

Implementing safe versions of POSIX interfaces with more than one pathname (i.e., `safe-rename` and `safe-link`) can be problematic on systems that don't support `renameat` and `linkat`. The emulation of these functions is complicated by the fact that a process can have only one current working directory at a time; as a workaround one could perhaps utilize temporary directories with random names as intermediaries.

Current POSIX standards still lack some primitives that operate on existing files by file handle instead of file name, but this may change as standards evolve. For example, the recently-standardized `O_EXEC` (open file for execute) flag [ope08] enables the implementation of a family of `fexec` primitives that execute the file specified by a file handle.Support for these is already implemented in some Linux and BSD versions. Based on these primitives one could implement `safe-exec` versions that can recover from accessing an unexpected file, similar in the way that `safe-open` recovers before performing an irreversible operation. We note that executing files in unsafe directories is a minefield, and leave the development of a suitable safety policy as future work.

6.5 Experimental validation

We conducted extensive experiments to validate our approach for safe pathname resolution. Our goals in these experiments were (a) to check whether existing applications would continue to work when they run over a POSIX interface that implements safe pathname resolution; and (b) to see if we can identify yet-undiscovered vulnerabilities related to applications that follow unsafe links.

6.5.1 Testing apparatus

We implemented our safe name resolution and tested several "live" systems, to see what applications actually use unsafe links, and for what purpose. To cover a wide range of operating

systems and production environments, we opted for implementing our procedure in a "shim" layer between the applications and `libc`. That is, we built a library that intercepts filesystem calls, and instructed the run-time linker to load it before the regular `libc`. We used this to instrument dynamically-linked programs including setuid and setgid programs.While the LD_PRELOAD environment variable was sufficient to instrument most programs, instrumenting setuid and setgid programs required additional steps. We stored run-time linker instructions in `/etc/ld.so.preload` on Linux, and in `/var/ld/ld.config` on Solaris; we modified the run-time linker `/libexec/ld-elf.so.1` on FreeBSD. This approach makes it easier to test existing systems, but it may not be able to interpose on calls between functions within the same library. In addition it is necessary to intercept some library calls not related to file access, to prevent the accidental destruction of environment variables or file handles that our "shim" layer depends on.

In the interposition library, we implemented the safe pathname resolution and used it in the filesystem calls `open`, `fopen`, `creat`, `unlink`, `remove`, `mkdir`, `rmdir`, `link`, `rename`, `chmod`, `chown`, and the `exec` family. With `openat` and related functions, we did not implement yet safe pathname resolution with respect to arbitrary directory handles; in our measurements, such calls were a tiny minority. So far we only instrumented calls that involve absolute pathnames, or pathname lookups relative to the current directory.

We also kept some state related to the current working directory in our library, in order to implement safe name resolution for relative pathnames. (The same approach can be used for the directory handles used by `openat` and related functions, but we did not implement this yet.) A more detailed description of the implementation and its intricacies is provided in Section 6.8.

6.5.2 Measurements of UNIX systems

We ran our pathname safety measurements on several out-of-the-box UNIX systems, specifically Fedora Core 11, Ubuntu 9.04, and FreeBSD 7.2 for i386 (both server and desktop versions). These systems were run on VMware workstation 5 for Linux and Windows hosts, and on real hardware. We instrumented the top-level system start-up and shutdown scripts, typically `/etc/rc.d/rc` or `/etc/init.d/rc`, and were able to monitor system and network daemon processes as well as desktop processes.For this instrumentation, we disabled security software such as AppArmor and SELinux to avoid interference between our instrumentation and their enhanced security policies. In all of these experiments, we configured our library to run in a report-only mode, where policy violations are logged but the intended operation is not aborted. (In fact, following the complete pathname resolution, our library will simply make the underlying system call on the original arguments and return the result.)

We ran these systems in their out-of-the-box configurations, and also tested some applications including the Gnome desktop, browsing with several Firefox versions (including plugins for popular multi-media formats), office document browsing, printing with Adobe Acroread, software compilation with gcc, and software package installation. The vast majority of these tests passed without a hitch. Most systems and applications never attempted

an operation that would violate our safety policy, and thus they would have worked just as well had we configured our safe name resolution in enforcing mode. One notable exception is the web-server application, discussed in Section 6.5.5.

6.5.3 Latent vulnerabilities

In the course of our experiments we uncovered a number of latent privilege escalation vulnerabilities. The latent vulnerabilities occur where privileged system processes write or create files as `root` in directories that are writable by an unprivileged process. In these cases, a compromise of an unprivileged process could result in further privilege escalation:

- The Common UNIX Printing System (CUPS) saves state in files `job.cache` and `remote.cache`. These files are then opened with `root` privileges and with open flags `O_WRONLY|O_CREAT|O_TRUNC`, in directory `/var/cache/cups` which is writable by group `lp` (on some systems group `cups`). The CUPS software uses this group when running unprivileged helper processes for printing, notification, and more. If an unprivileged process is corrupted, an attacker could replace the state files by hard or symbolic links and destroy or corrupt a sensitive file.

- Similarly, a latent problem exists with files under directory `/var/log/cups` on Fedora Core 11.

- During MySQL startup, the `mysqld` daemon opens a file *`hostname`*`.lower-test` with flags `O_RDWR|O_CREAT` as `root`, under directory `/var/lib/mysql` which is owned by the `mysql` user. If the `mysqld` daemon is corrupted later when it runs with user `mysql` privileges, an attacker could replace this file by a hard or symbolic link and corrupt a sensitive file when MySQL is restarted.

- The Hardware Abstraction Layer daemon opens a file with flags `O_RDWR|O_CREAT` as `root`, in directory `/var/run/hald`. This directory is owned by user `haldaemon`, who also owns several daemon processes. Some of these processes listen on a socket that is accessible to local users.

- The Tomcat subsystem opens a file with flags `O_WRONLY|O_APPEND|O_CREAT` as `root` in directory `/var/cache/tomcat6`. This directory is owned by user `tomcat6`, who also owns a process that provides service to remote network clients.

- On Fedora Core 11, directory `/var/lock` is writable by group `lock`, which is also the group of a setgid program `/usr/sbin/lockdev`. System start-up scripts create "lock" files as `root` with flags `O_WRONLY|O_NONBLOCK|O_CREAT|O_NOCTTY`. If the `lockdev` program has a vulnerability, an attacker could replace a lock file by a hard or symbolic link and corrupt a sensitive file.

- XAMPP [K.S] (an integrated package of Apache, MySQL, PHP and other components) on Linux opens files, for error logging, as `root` in the directory `/opt/lampp/var/mysql`

which is owned by the uid `nobody`. A corrupted process running as `nobody` can replace this with a link to any file on the system which would then be overwritten. We note that XAMPP runs a number of daemons providing network services as the `nobody` user, including `httpd`.

In all these cases, our safe name resolution would protect the system from privilege escalation if the unprivileged processes are corrupted.

6.5.4 Policy violations

During our "whole system" tests we ran into a surprisingly small number of actual safety policy violations. These turned out to be specific to particular platforms, and were caused by quirks in the way that directory ownership and permissions were set up:

- On FreeBSD 7.2, the `man` command could trigger policy violations when a user requested a manual page. FreeBSD stores pre-formatted manual pages under directories owned by user `man` (instead of `root` as with many other UNIX systems). According to our policy, these directories are unsafe for users other than `man`. This resulted in policy violations with pre-formatted manual page files that had multiple hard links.

  FreeBSD adopted this approach so that pre-formatted manual pages can be maintained by a non-`root` process. This limits the impact of vulnerabilities in document-formatting software. We find the benefits of this approach dubious: document formatting software still runs with `root` privileges when the super-user requests a manual page for software that is not part of the base system. By default, no pre-formatted manual pages exist for this software category, and this is where the biggest risk would be.

- The FreeBSD package manager triggered warnings about following '`..`' when removing a temporary directory tree under `/var/tmp`; these could be addressed by a more permissive policy (cf. Section 6.6.1).

- On Fedora Core 11, the Gnome desktop software triggered policy violations that we did not experience with other systems. The violations happened when a process with `gdm` user and group privileges attempted to follow symbolic links under directory /var/lib/gdm. This directory is writable by both owner `gdm` and group `gdm`.

  These policy violations can be avoided with a more sane configuration that uses owner `gdm` write permission only. Our "live" measurements show that group `gdm` is used only by processes that run as user `gdm`. With a single-member group like `gdm`, owner `gdm` permission is sufficient, and group `gdm` write permission is unnecessary. (We found similar issues with XAMPP for Linux, which installs with directories that have owner `nobody` and group `root` with group write permission.)

6.5.5 A web-server application

Most of our measurements were done on bare-bones systems that we instantiated specifically for the purpose of running the experiments. The only production system that we had access to was a Debian 5.0 system running an Apache web server and some other services. On that system we did not attempt a whole-system measurement, but instead only run specific services under our measurement apparatus. Also on that system most services did not report any policy violations, with the notable exception of the web server.

The web site on that system is managed cooperatively by several users, where different users are responsible for different parts of the site, and with no attempt for any protection between these users. As a result, the web-tree is a mesh of directories with different owners, many of them writable by the `web-administrator` group (whose members include all these different users). Roughly speaking, the entire web-tree on that system is an UNsafe subtree. Moreover, some dynamic-content parts of the web site make heavy use of symbolic links, e.g., for using the same script in different contexts.

It is clear that our `safe-open` procedure will break this web site, but this is more an artifact of our particular choice of implementation than of the security guarantee that we set out to ensure. Indeed, in Section 6.6.1 we describe a more permissive implementation of `safe-open` that still ensures the same security guarantee, but would not break this web site. (The idea is that we can follow symbolic links off unsafe directories, as long as we ensure that the file that we get to at the end does not have any safe names.)

6.5.6 Conclusions

Our experiments seem to indicate that our approach to safe name resolution is both effective and realistic. On one hand, it fixes *all 177 symlink-related vulnerabilities reported in CVE since January 2008*, and also provides protection against the (latent) vulnerabilities that we identified in our experiments. On the other hand, most systems will continue working without a problem even if this safety measure was implemented. The few that break can be "fixed" either by implementing a more standard permission structure for the relevant directories or by implementing the more permissive variant of `safe-open` from Section 6.6.1.

We stress that in our experiments, we did not identify even a single example where there is a legitimate need to open files that would be inherently disallowed by our approach to safe name resolution.

6.6 Variations and Extensions

6.6.1 A more permissive `safe-open`

Our `safe-open` procedure does not follow symbolic links off an unsafe directory, but is not hard to see that this policy is more restrictive than what we really need for our security guarantee. Indeed, we only need to ensure that `safe-open` fails on an unsafe name *if the file to be opened has any other name that is safe*. It turns out that a small modification of

`safe-open` can ensure the same security guarantee while allowing more names to be opened.

The idea is to keep two safe/unsafe flags rather than one. Both flags begin in a *safe* state and switch to *unsafe* state when visiting an unsafe directory, but one flag is "sticky", in that once in *unsafe* state it stays in this state until the end of the name resolution, while the other is reset to the `safe` state whenever we are about to follow a symbolic link with an absolute path. That is, the second flag is reset to `safe` state whenever we are about to return to the root directory.

With these two flags, we can follow arbitrary symbolic links, and can also follow '`..`' as long as the second flag is in *safe* mode. When we finally reach the file to be opened, we abort the procedure only if (a) the "sticky" flag is in *unsafe* mode and the file has more than one hard link, or (b) the two flags have different values. (In the second case, the "sticky" flag indicates that the given pathname was unsafe, while the resettable flag indicates that as part of the name resolution we followed some safe name to arrive at the file.) The reason that this more permissive procedure works, is that if a file with only one hard link has any safe names, then its "canonical" name (i.e., the one with no symbolic links) must be safe. Moreover, this name must be the one followed by the time that the name-resolution arrives at the file itself.

As we described it, this more permissive version still refuses to follow '`..`' when the second flag is in *unsafe* mode. This can be easily remedied, however: we simply drop the restriction on following '`..`', and instead just reset the second flag to *safe* mode after every '`..`'.

6.6.2 An alternative `safe-open` using extended attributes

On some systems, a much more direct approach is also possible. Recall that the problem that we try to address is that an adversary without permissions to a file is able to add names to the filesystem that resolve to that file. If the filesystem supports extended attributes, then we can avoid this problem simply by including with the file an attribute that lists all the permissible names for that file. The `open` procedure, after opening the file, will look for this extended attribute, and if found it will compare its pathname argument against the list of permissible names, and will abort if there is a mismatch. For example, the file `sudo` in `/etc/init.d/` will have a permitted-names attribute listing the names `/etc/init.d/sudo` and `/etc/rcS.d/S75sudo`, and no program will ever be able to open it using any other name.

This simple solution looks quite attractive, but it necessitates proper management of the additional attribute. In particular, we must decide who may set this attribute (and under what conditions). For example, when we add to our filesystem a symbolic link: do we need to modify the permitted-name attribute in all the files under `/var/mail/`? We leave all these questions to future work.

6.6.3 Group permissions

Recall that our `safe-open` procedure only uses `uid`s to determine safety of directories, which means in particular that we treat two processes with the same `uid` as equal and do not try to protect one from the other. This leaves open the possibility of privilege escalation by acquiring group privileges: namely, an adversarial process may try to trick another process with the same effective `uid` but more group privileges into opening a file that the adversarial process itself cannot open.

In the work we do not try to protect against such attacks, indeed protection between different processes with the same effective `uid` is virtually impossible in most POSIX systems. We mention that it is not hard to change the `safe-open` procedure itself so that it considers the `gid` rather than the `uid` for the purpose of determining directory safety, but this would require a change in the interface, since the calling application would need to somehow indicate that it wants to use this `gid`-based safety check instead of the default `uid`-based check.

We note that our approach for safe name resolution is quite coarse with respect to group permissions, in that group write permissions always make a directory unsafe for everyone. This is justified when the directory `gid` is the primary or secondary `gid` of multiple UNIX accounts, since multiple accounts are manipulators. However, contemporary UNIX-es have many `gid`s that are associated with only one `uid` (or maybe none at all, e.g. when the `gid` is only used by the execution of a setgid program). In general we cannot anticipate all possible ways that a `gid` may be activated, and hence we consider the directory unsafe in all these cases. This may trigger spurious policy violations in some configurations, but in our experiments we did not find configurations where such policy violations cannot be resolved.

We also note that in conjunction with the more permissive variant from above, this behavior lets administrators bypass much of our safety mechanisms: To forgo most of our safety protections for some subtree (without otherwise changing any permissions), it is sufficient to make the root of that subtree writable, e.g., by the `root` group. Assuming that only `root` is a member of this group, this will not change any real permissions in the system, but will make that entire subtree unsafe, and therefore permit opening of the files in it also using other unsafe names, even ones with symbolic links. (This trick does not help if there are multiple hardlinks, however.)

6.7 Relative pathnames

When resolving a pathname relative to an initial directory (i.e. the current directory or a directory handle with functions such as `openat`), the resolver needs to determine if the initial directory is safe, before following the same steps as with absolute pathnames (Section 6.2.2). For this, the implementation needs to maintain safety information about directory handles, including the implicit directory handles for the current and root directories of all processes.

The per-handle safety information needs to be initialized when a directory handle is instantiated with functions such as `open`, `chdir` or `chroot`, and the safety information needs to be propagated when a directory handle is copied with functions such as `dup`, `fcntl`, `fork`,

or with functions that transmit a file handle over an inter-process communication channel. Maintaining this information is straightforward in the file system itself (i.e. in the operating system kernel). We discuss our user-level approach in Section 6.8.

In a simplistic implementation, each directory handle has a static flag that indicates if the directory is safe. However, additional care is needed with processes that change their effective `uid` (for example, a process that invokes the `seteuid` function, or a process that executes a file with the setuid bit turned on). As the result of an effective `uid` change, a directory that was safe may become unsafe or vice versa. As a further complication, the safety of a directory depends on the program execution history. For example, a handle for directory `/etc` is normally safe for everyone, but that same directory handle would be safe only for `joe` if a pathname resolved through a symbolic link under `/home/joe`.

To account for processes that change execution privilege, we propose that each directory handle would include a field specifying the `uid` that the directory's pathname prefix was "safe for" when the pathname was resolved. Namely, this field will indicate `root` if the directory was reached via a system-safe pathname, it will indicate a single non-root `uid` if it was reached via a pathname that has only `uid` and `root` as manipulators, and it will indicate *no-one* if the pathname had more than one non-root manipulator. When resolving a pathname relative to an initial directory, one determines the safety of the initial directory by combining the "safe for" `uid` from the handle with fresh information about the owner and writers for the initial directory itself.

6.8 User-level implementation

As mentioned earlier, a kernel-based implementation of safe pathname resolution is straightforward: while visiting each pathname element one at a time, maintain a safety flag and apply the safety policy for following symbolic links, "`..`", and for files with multiple hard links as appropriate. With a kernel-based implementation, maintaining per-handle directory safety information is also straightforward. This approach is preferable, but only after it has been demonstrated that safe pathname resolution does not break well-behaved programs.

To demonstrate the feasibility of our pathname safety policy, we chose an approach that is based on library-call interposition with an in-process monitor. This approach works with dynamically-linked programs, including programs that are setuid or setgid, and it provides acceptable performance on Linux, FreeBSD and Solaris systems. We opted against external-process monitors such as `strace` or `truss`: they suffer from TOCTOU problems, they cause considerable run-time overhead, and they don't have direct access to the monitored process's effective `uid` which is needed for pathname safety decisions.

As illustrated in figure 7, the monitor is implemented as a library module that is loaded into the process address space between the application and the libraries that are dynamically linked into the application. Depending on configuration, the monitor can log function calls such as `open` with the effective `uid`, and can log whether or not a call violates our pathname safety policy. For the purpose of the feasibility test the monitor does not enforce policy, but instead passes control to the real `open` etc. function. The in-process monitor for Linux,

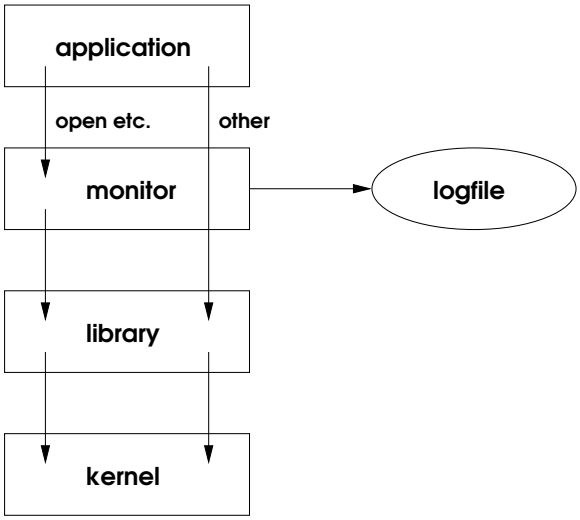


Figure 7: In-process monitor architecture.

FreeBSD and Solaris is implemented in about 2000 lines of K&R-formatted C code, comments not included, plus a small shell script that implements the command-line interface.

Besides interposing on functions such as `open` that require pathname resolution, our in-process monitor interposes on additional functions to ensure proper operation of the monitor itself. For example, the monitor intercepts function calls such as `close` and `closefrom`, to prevent the logging file handle from being closed by accident. The monitor intercepts function calls such as `execve` to ensure consistent process monitoring when a new program is executed. Upon `execve` entry, the monitor exports environment variables to control run-time linker behavior and to propagate monitor state, and it resets the close-on-exec flag on the logging file handle. When the `execve` call returns in the newly-loaded program, the monitor restores private state from environment variables before the application's code starts execution.

To track per-handle directory safety state, an in-process monitor would need to interpose on functions that copy file handles such as `dup` or `fcntl`. Interposition is not necessary with process-creating primitives such as `fork` or `vfork`, since these are not designed to share the in-kernel file descriptor table or process memory between parent and child processes. On the other hand, the Linux `clone` and BSD `rfork` process-creating primitives are designed so that they can share the file descriptor table or process memory, meaning that changes made by one process will affect the other process. This behavior complicates a user-level monitor implementation, and is not yet supported by our monitor.

Our preliminary in-process monitor maintains a safe/unsafe flag for directory handles created with `open` so that it can check programs that use the `open-fchdir` idiom. The monitor does not yet check lookups relative to a directory handle. In our measurements, we found that the `openat` etc. functions are used by only few programs, and that those functions are called almost exclusively with absolute pathnames or with pathnames relative to the current directory. The monitor currently does not propagate the per-process current

directory and root directory safety state across function calls such as `execve`. Instead, it initializes their safety state on the fly at program start-up time. Without modification to monitored applications, it is not practical for a user-level monitor to track safety flags for directory handles that are sent over an inter-process communication channel. Fortunately, such usage is rare.

6.9 Summary

In this section we considered the problem of privilege escalation via manipulation of filesystem pathnames, which effect name resolution in system calls such as `open`, `unlink`, etc. While many privileged programs take measures to protect against such attacks, these measures are always very application specific. We propose a more general approach of having safe pathname resolution as part of the filesystem itself or a system library, thereby protecting all applications by default.

We introduced the concept of the *manipulators* of a pathname, that include anyone who can influence the outcome of the pathname resolution. In POSIX these are the users who either own or can write in any directory visited during the pathname resolution. Using this concept, we call a pathname *safe* for U if the only manipulators of the pathname are `root` and U. We described a general routine `safe-open`, ensuring that if a file has safe names then `safe-open` will not open that file with an *unsafe* name, and demonstrated that this guarantee can be used to thwart filename-based privilege escalation attacks. This is useful not only for privileged programs that run in known-to-be hostile environments, but also for programs written by naive developers, and programs that are being deployed in unforeseen environments with unexpected file permission semantics.

We implemented our safe name resolution routine in a library, using portable code over the POSIX interface, and performed extensive experiments to validate the applicability of our solution to current operating systems and applications. We verified that this solution uniformly protects system against the documented cases of applications and daemons vulnerable to pathname manipulation attacks, as well as against some new (latent) vulnerabilities that we uncovered. We also instrumented current versions of Ubuntu 9.04, Fedora Core 11 and FreeBSD 7.2 to run every process through a program which interposes calls to file manipulation and related calls and checks if the corresponding operation manipulates a safe pathname. These experiments confirmed that very few existing systems break when used over our safe name resolution, and the handful of cases where our solution produces false positives can be handled either by implementing a more standard permission structure for the relevant directories or by using a more permissive variant of our solution.

7 IsoVisor:Secure Virtualization

Virtualization has changed the computing landscape from large-scale public cloud infrastructures to small-scale personal devices, with benefits ranging from improved resource utilization to improved deployment flexibility. The key underlying primitive that allows virtualization to be realized is the transparent sharing of hardware resources by multiplexing virtual-machine requests and demultiplexing hardware responses. In effect this makes a virtual machine (VM) believe it has exclusive ownership and control over the hardware, all the while the virtualization platform underneath allows multiples VMs to access the shared hardware resources concurrently. A side effect of virtualization is that the transparent sharing of hardware resources seems like an attractive way to achieve isolation between VMs. If each VM can operate only as if it is the only workload on the hardware platform, then it is effectively isolated from other VMs, which is especially important when different VMs belong to different customers, to administrative domains at different security levels in an organization, or to different tiers of a multi-tier application.

Unfortunately, it turns out that virtualization's transparent sharing does not provide a strong level of isolation between VMs. Research has shown that VM co-location can introduce opportunities for "cross talk" due to hypervisor vulnerabilities [Fer07, Orm07] and side channels that leak information about a virtual machine's CPU load, cache accesses, and interrupts [RTSS09, ZJOR11]. Even in non-virtualized environments, side-channel attacks have been used to transfer cryptographic keys between processes sharing the same CPU [Per05, WL06, KASZ08, AcKKS07, AHFG10, TOS10, ARJS07, Bel05]. Although no attacks have been reported yet that transfer cryptographic keys between virtual machines, we expect that these will eventually become possible. Thus, the transparent-sharing property exhibited by virtualization platforms is insufficient to guarantee isolation.

Distributed virtualized computing resources can be seen as built up by modularly adding new virtualized resources such as storage, databases, services, and other applications. This directly lends itself very to the idea of analyzing them modularly and aiming for security guarantees that are composable. Thus this is a very natural candidate for analysis with the Universal Composability framework. Our goal is to write down a formal model of isolation and using the UC Framework to establish additional properties to be satisfied by a virtualization platform and proving their sufficiency for isolation.

Existing security models of isolation cannot be easily translated into specifications for implementing a virtualization platform. For example, non-interference and separability, although sound security models, are defined on abstract machines, making it quite challenging to apply them to real systems. Contemporary hardware goes beyond CPU, memory, and storage, to include a large number of peripheral devices, communication buses, caches, and memories, each of these with particular interfaces, particular access controls, and particular operational latencies. Modern hypervisors, which would need to satisfy non-interference for example, contain significant amounts of code to configure, schedule, and manage the hardware components. We address this challenge by targeting our requirements to concrete software and hardware components in modern virtualization platforms.

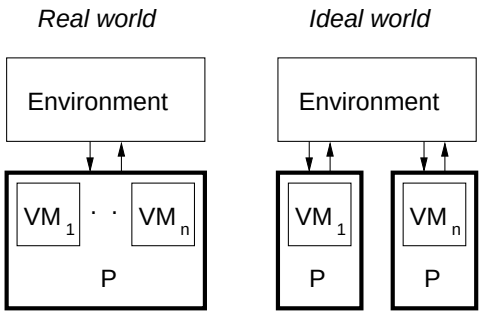


Figure 8: Correspondence between real-world and ideal-world execution. Workloads execute in virtual machines VM, on hypervisor and hardware platforms P; hardware boundaries are shown as heavy lines. In the real world (left) different virtual machines share the same platform. In the ideal world (right) each virtual machine executes on a dedicated platform.

Our overall approach is illustrated with Figure 8. Intuitively, we wish to achieve in the real world of shared hypervisor and hardware resources an isolation level that is equivalent to that of an ideal world where no resources are shared. First, we present a formal model for isolation in virtualization platforms, which (roughly) ensures that a workload running in one virtual machine cannot infer or affect the private state of workloads in other virtual machines except through the external environment.

Next, we derive five requirements (called Loc Separation, Implicit Parameter Separation, Error Signaling Separation, Conf Separation, and Next Call Separation) which guarantee isolation when satisfied by a virtualization platform. The five requirements correspond respectively to the absence of explicit information flows, the independence of state of a VM from the state of any other VM, the independence of errors seen by a VM from the state of any other VM, the absence of implicit (timing-based) information flows, and the independence of execution of a VM from the state of any other VM. To prove that these requirements are sufficient, we use Universal Composability to show the equivalence of isolation in an ideal cloud (i.e., non-shared hypervisors and hardware) and a real cloud (with shared hypervisors and hardware) that satisfies these requirements. Finally we study Loc Separation and Conf Separation in detail and translate them into practical properties for virtualization-platform implementations.

Underpinning our approach there is a *proof-driven strategy* that focuses on the abstract security properties ensuring that virtual machines are free from entire classes of direct and side-channel leaks. In contrast, attack-driven approaches defend against specific leaks, for example by eliminating hypervisor, cache, or CPU sharing [KSRL10, RNSE09, ZJOR11].

In summary, we make the following contributions:

- We formalize the "physically separate hardware" isolation model (called *noLeak*) for virtualization platforms, capturing both explicit and implicit (timing-based) information flows, and derive five requirements for real-world, hardware-sharing implementations to achieve equivalent isolation. Furthermore we formally prove in the Universal Composability framework that these five requirements are sufficient for isolation.

- We provide what we believe to be the first formal treatment of timing-based side channels that goes to the root cause of the problem, the scheduling policies of hardware and software resource arbiters. We analyze both deterministic scheduling policies (for which we derive an isolation proof) and probabilistic scheduling policies (for which we define and quantify information leakage rates in certain stateless resource arbiters).

- We analyze the root cause of explicit information flows and demonstrate that access control (via address translation or access matrices) and correct execution of context save and restore operations is sufficient to prevent explicit information flows in commodity virtualized platforms. Our results cover major virtualization technologies, including software emulation technologies such as dynamic binary translation and paravirtualization, CPU-supported virtualization, self-virtualizing peripheral devices, and IOMMU.

- We analyze the root causes of storage side channels and derive distinct requirements for the isolation of storage side channels induced due to error conditions and due to sharing of internal platform state.

7.1 Related Work

The concept of isolation between principals has been studied under various aspects since the beginnings of computer security. Various models from Bell La Padula to Clark-Wilson to their many derivatives provide different definitions of what it means for a system to be secure by protecting the confidentiality and integrity of its principals' data. While it is not possible to provide a complete overview of the space of formal security models (see Mantel for a survey [Man03]), we mention several models that are closely related to our isolation model.

Models of isolation. Goguen and Meseguer introduced non-interference [GM82] as a security model built on Denning's secure information flow model [Den76]. Non-interference defines security as preventing high-privilege data to flow to low-privilege sinks, and thus could provide a foundation for our definition of isolation. Later extensions of non-interference, such as generalized non-interference [McC87] and double generalized non-interference (DGNI), are related to our isolation formalism. DGNI for example requires that the independence relation between high-privilege principals and low-privilege principals is symmetric, just as we require. McLean's separability restricts non-interference to require that a system with both high-privilege and low-privilege principals can be viewed as composed of two systems, one with only high-privilege principals and one with only low-privilege principals [McL94]. To the best of our knowledge, ours is the first model that maps a high-level property (i.e., isolation) onto a practical hypervisor-based architecture by specifying requirements for the hardware- and hypervisor-based resource management and access-control components.

In the context of cryptographic systems, Backes and Pfitzmann introduced a definition of probabilistic non-interference that reflects the dependency on the computational assumptions of the underlying cryptographic primitives [BP02, BP03, BP04]. In contrast, we consider a

probabilistic definition of resource arbiters in general-purpose systems and provide a way to compute the leakage rate for such cases.

Proofs of isolation. Unfortunately formal proofs of isolation are rare for practical systems. When they do exist, the proofs show that a low-level implementation of a system satisfies its high-level specification, but do not establish the security properties of the specification itself. Examples include the seL4 microkernel work by Klein *et al.* [KEH+09], which presents the experience of formally proving the satisfaction of an abstract specification of safety and deterministic behavior properties, and the NOVA micro-hypervisor work by Tews *et al.* [TWV+08], which describes a limited-scope verification using a formal description of IA32 hardware and memory mapped devices. In contrast, we focus both on a natural definition of isolation (i.e., the desirable security property) and a separate set of requirements (i.e., the high-level specification), and establish provably sufficient conditions for their equivalence.

Our work is most closely related to separation kernels [Rus82, KZB+91, WSG02, Nat95, KEH+09, Gre, SK10]. Rushby [Rus82] provides a proof of separability and derives sufficiency conditions for an abstract separation kernel, but excludes covert channels and denial of service and provides no implementation. In particular, one of our requirements (Loc Separation) can be seen as a logical equivalent of Rushby's definition. Our work provides a proof of isolation with sufficiency conditions for a general model of shared virtualization platforms for both direct and side channels.

Covert and side channels. Covert channels and their countermeasures have been studied since the early days of trusted computing [Lam73, Lip75, Kem02, GM82, KW91, Wra91]. Recently, side channel attacks have received increased attention. One class of these take advantage of the execution latencies in multicore and hyper-threaded CPUs that are caused by shared micro-architectural elements such as functional units, caches and branch prediction units [Per05, WL06, KASZ08, AcKKS07, AHFG10]. To the best of our knowledge, we are the first to model formally the root cause of side channels present in execution latencies, in terms of the shared internal state of platform-level resource managers.

Mitigating side channels in an existing system or building a new system free of side channels has proved to be a hard problem. Many proposed solutions rely on a fixed partitioning of the hardware resources [KSRL10, RNSE09], on placing VMs on separate hardware [RNSE09], or on using the side channel itself to detect unfriendly placement of VMs [ZJOR11]. Köpf provides a formal approach to addressing side channels [Köp07], and his work is closest to our present formalization effort. Similar to Köpf, we focus on information flow through observable patterns of requests and grants, but we abstract away the parameters of requests, thus obtaining a more specific notion of timing side channel, and we only consider non-adaptive adversaries. There are key differences that we believe make our approach more practical. We do not constrain the structure of programs executed on the platform, while preserving the sufficiency of our isolation requirements. Thus, the onus of constructing leakage-free programs and VMs is shifted to constructing leakage-free virtualization platforms. Additionally, we are able to establish necessary structural conditions, such as being time and space multiplexed, on the resource-management components in our setting.

7.2 Threat model

In this section we try to capture in some detail the precise problem we are attempting to solve, the assumed attacker model, and the implicit trust assumptions.

7.2.1 Problem Statement

Our target environment is a system consisting of a hardware platform, a hypervisor software component, and one or more virtual machine software components. The hypervisor could be a Type I or "bare-metal" hypervisor, or a Type II or a hosted hypervisor. The hardware provides facilities to store, retrieve, and perform computation with data, as well as to interact with any number of peripherals for networking, external storage, etc. The hypervisor controls the access of the virtual machines to the hardware platform. The virtual machines run arbitrary software that performs private computations over sensitive data and that makes requests to the hypervisor when it needs to interact with the hardware platform. The problem of isolation is for the hypervisor to ensure that software in one virtual machine cannot infer or influence the private state or computation in another virtual machine via their shared use of the hypervisor and the hardware. We deductively derive requirements to achieve isolation.

We do not consider virtualization that is implemented primarily in software such as OS API virtualization (e.g., Solaris zones [PT04], AIX WPAR [Mil08], Linux Vserver [* *08], FreeBSD jails [KW00]), or CPU emulation (Java [Ora11b], QEMU [Bel05]), although our framework could be applied in those settings, with the appropriate assumptions (e.g., in OS API virtualization the TCB consists of anything underneath the OS API, including the OS and the hardware).

7.2.2 Trust Assumptions

We assume that the hardware exposes a well-defined instruction set architecture (ISA) for each CPU present and a well-defined API for each peripheral present, where "well-defined" means that the hardware platform correctly implements a known functional specification. Furthermore we assume the hardware platform has functionality for setting access-control permissions for particular components of the hardware platform. This access control can be set such that only the hypervisor code, as defined at boot time, can access those components and can change the access-control permissions.

7.2.3 Attacker Model

The attacker controls one or more virtual machines, and has full capability within the virtual machine(s) it controls, can perform arbitrary computation, and can make any number of requests to the hypervisor. An attacker's purpose is to infer or modify private state of other virtual machines not under his control. Since all workloads execute in virtual machines, we assume the presence of a hypervisor is known and do not concern ourselves with hiding the hypervisor [GAWF07]. We focus on explicit information flows that are in direct violation

of hypervisor isolation policies, and on storage-based and timing-based implicit information flows that result from execution on shared resources.

7.2.4 Physical-channel exclusions

We assume that the system is physically secure. We therefore exclude hardware-level attacks, such as attacks that give access to current or previous storage content [HSH+08, Gut96] or that induce hardware errors with radiation or other such mechanisms [GA03]. Similarly, we also exclude attacks that exploit collocation-enabled via non-digital channels such as power consumption [KJJ99], emissions of electromagnetic, optical, acoustic or other nature [AARR02, Kuh02, ST04], or temperature drift [ZBA10]. We make the assumption that virtual machines have no direct access to such non-digital channels.

7.3 Requirements for Isolation

This section presents a definition of isolation based on the equivalence of VM execution on dedicated and shared platforms. We then deductively derive a set of requirements for equivalence of execution based on a model for execution on a stored program computer.

7.3.1 Isolation

As captured in Figure 8 our notion of isolation for virtual machines executing on a *shared* platform (real world), consisting of a type I or type II hypervisor and hardware, is equivalent to that of virtual machines on *separate* platforms (ideal world). Thus there is an *ideal world* in which VM are naturally isolated on separate platforms, and a *real world* in which virtual machines share a platform. Informally, we say that isolation is achieved when through its interactions with the platform and the environment, no virtual machine (whether malicious or benign) is able to distinguish whether it is executing in the real world or ideal world. This requires that all interactions of a VM with the platform and environment provide the same outputs in the ideal world and in the real world. In Section 7.4, we formalize this notion of equivalence using the Universal Composability (UC) framework.

7.3.2 Platform Model

A platform comprising a type I or type II hypervisor and hardware emulates a stored program computer. We introduce a simple model of a stored program computer to capture the execution resources available to VM. To model execution latencies which can give rise to timing side channels, our model includes a global clock.

7.3.2.1 Processing Element A processing element (PE) is an entity that performs computations. This is an abstraction for hardware and software entities that perform computation such as CPU and disk controllers, virtual devices, hyper-calls, and system calls.

7.3.2.2 Interface Element An interface device communicates with the environment by sending and receiving bit strings. The environment is everything other than the platform and the VM.

Note that a hardware or software device must be partitioned into a processing element and an interface element under this model. The interface element subsumes the functionality that is used to communicate with the environment, and the processing element includes all other functionality.

7.3.2.3 Memory Memory is an execution resource that can store bit strings. We model memory as a set of *locations*. This is an abstraction for architecturally-visible memory such as main memory, CPU and device registers, and disk sectors.

7.3.2.4 Global Clock We model the global clock as a free-running counter that increments with a period equal to the latency of the fastest operation of the computer system. For convenience of exposition, we assume that the global clock is synchronized to a global time source.

7.3.3 Platform Interface for Virtual Machines

Using the execution resources of the platform, we define a *platform interface* that VMs use to interact with the platform. The goal of our formalism is to prove isolation against this interface, i.e., rather than state specific attacker models, as the the capabilities of the attacker are captured in the platform interface. This allows us to prove the isolation for *any* system where VMs (both good and malicious) use this interface.

In this interface, each invocation has an effect on the underlying *machine state* defined as follows:

Definition 8 (Machine State) *The platform has a set of storage containers in hardware for bit strings. At any time t, machine state $\sigma(t)$, capturing the contents of all the storage containers, is defined as a set of tuples $(storage\ container, bit\ string)$.*

When it is clear which time instant we are referring to, we will drop the functional dependence on t.

The interface captures two disjoint sets of interactions: those with the platform, and those with the environment. The first set of interface calls we define are simply an abstraction of the load operand–execute–store result execution paradigm of a stored program computer. Given that the entity implementing the platform interface is a simulator for the hypervisor and hardware, we define three interface calls $ReadLoc$, $RequestPE$, and $WriteLoc$ to capture the functions of load operand, execute, and store result respectively. All calls update the machine state σ, and return errors when appropriate. Also implicit in the state change is the advance in the global clock by an amount equivalent to the latency of the operation.

$ReadLoc(VM_i, loc)$: Request to the platform by virtual machine VM_i to read memory at location loc.

$WriteLoc(VM_i, loc, val)$: Request by virtual machine VM_i to write val to the memory location loc.

$RequestPE(VM_i, PE, I)$: Request by virtual machine VM_i to perform operation I on processing element PE. The platform has loaded the operands of I as a result of a sequence of $ReadLoc$ interface calls.

$ReadTimer(VM_i)$: Request by virtual machine VM_i to read the global clock.

The remaining calls model communication between the VM and the environment. As before, all calls update the machine state σ, return an error instead of the requested result as appropriate, and return after the global clock has advanced by an amount equal to the latency of the operation.

$Send(VM_i, IE_i, val)$: Request by virtual machine VM_i to interface element IE_i to send bit string val to the environment.

$Receive(VM_i, IE_i)$: Request by virtual machine VM_i to interface element IE_i to receive a bit string from the environment.

We assume that in both the ideal and real worlds, the VM only receives messages addressed explicitly to it.

7.3.4 Deriving Requirements for Isolation

This section identifies conditions which ensure that a VM can not distinguish between the real and ideal worlds. Since a VM can only interact using the interface calls from Section 7.3.3, indistinguishability implies that the VM observes the same results upon invocation of any interface call. We observe that the result of an interface call depends only on the machine state σ since a VM must supply the same arguments to interface calls when executing in either world. Otherwise, the VM does not have a consistent basis for comparing the results obtained. A sufficient condition for isolation is the following notion.

Definition 9 (Observational Non-interference) *A virtualization platform satisfies observational non-interference if for each virtual machine VM_i running on that platform there exists a projection function π_{VM_i} of the machine state σ such that the effect of all interface calls made by VM_i depend only on the projection $\pi_{VM_i}(\sigma)$. More precisely, for all states σ and σ' with an equal projection, $\pi_{VM_i}(\sigma) = \pi_{VM_i}(\sigma')$, we have that:*

- *an interface call made by VM_i in state σ returns the same result as an interface call made by VM_i in state σ';*
- *if the updated states after a call by VM_i are $\tilde{\sigma}$ and $\tilde{\sigma}'$ respectively, then $\pi_{VM_i}(\tilde{\sigma}) = \pi_{VM_i}(\tilde{\sigma}')$; and*
- *for all $j \neq i$, $\pi_{VM_j}(\sigma) = \pi_{VM_j}(\tilde{\sigma})$. Similarly for σ'.*

Based on the definition of the interface calls, the results are of four types: (1) result of a computation, (2) input from the environment, (3) error signals, and (4) global time. We explore each in turn.

7.3.4.1 Interface Call Result For the read and write operations (*ReadLoc* and $WriteLoc$), in the ideal world, each VM works with its own copy of memory locations. Thus in the real world, equivalence to the ideal world yields the following requirement:

Requirement 12 (Loc Separation) *Let val be the quantity returned by* $ReadLoc(VM_i, loc)$ *at time t. If there was a previous write to loc by* VM_i*, then the immediately last write was* $Write(VM_i, loc, val)$*. Else if loc was initialized at the time of introduction of* VM_i *then it was initialized to the value val. If there was no previous write nor initialization, val is the constant indeterminate value* $\perp$*.*

$RequestPE(VM_i, PE, I)$ requests processing element PE to perform operation I. The input parameters to such a call are either from memory locations written to by the virtual machine VM_i addressed by the previous discussion on Loc Separation or they are *implicit parameters* derived from the internal state of the platform. We observe that the implicit parameters are updated as a result of operations performed by the platform comprising (1) operations arising from interface calls made by VMs, and (2) internal operations disjoint from operations used to service interface calls. Operationally, the values of the implicit parameters of a $RequestPE$ call are a function of the sequence of interface calls from the VMs and the sequence of internal operations of the platform. To provide observational non-interference, we require implicit parameters to be completely described by $\pi_{VM_i}(\sigma)$ for all VMs VM_i:

Requirement 13 (Implicit Parameter Separation) *For any virtual machine* VM_i*, processing element* PE*, operation* I*, and time t, if the interface call* $RequestPE(VM_i, PE, I)$ *at time t evaluates using an implicit parameter p, then the value of p depends on* $\pi_{VM_i}(\sigma)$ *only.*

7.3.4.2 Error Signals Errors signaled by interface calls are either due to hardware faults, which are outside the scope of this work (see Section 7.2), or those that depend on the input parameters of the interface calls and the internal platform state. Using an analysis similar to $RequestPE$ we state the following requirement:

Requirement 14 (Error Signaling Separation) *For all virtual machines* VM_i*, time t, and any interface call C by* VM_i*, the value of any error signaled depends on* $\pi_{VM_i}(\sigma)$ *only.*

7.3.4.3 Interface Call Latency The execution of any interface call requires execution resources from the platform. Each execution resource of the platform is associated with one or more entities that perform the arbitration between concurrent requests for that execution

resource. The arbitration functionality is normally implemented by the schedulers, allocators, and deallocators; examples include the instruction scheduler in an out-of-order CPU, the cache controller, and the VM scheduler within a hypervisor. Schedulers usually grant access to a resource for fixed time periods (e.g., a process scheduler grants access to a CPU for a fixed time slice), and resource managers grant access to a resource for an open-ended time period and then later revoke it (e.g., a cache manager grants access to cache lines and then revokes it by evicting the VM's data). Going forward, we will use the term *resource arbiter* to refer to both schedulers and resource managers and more generally to any entity that arbitrates between concurrent requests to execution resources.

Associated with every resource arbiter and request is an arbitration latency which is defined as the time that elapses from the instant the request is submitted to the arbiter to the instant the arbiter grants the resource requested. Note that the latency of an interface call is the sum of the execution latency of the sequence of operations required to evaluate the interface call and the arbitration latencies induced by all resource arbiters that grant execution resources to evaluate the operations of the interface call. The latency of evaluation of an interface call depends upon its input parameters and the resulting operations to be executed. Since the operations to be executed are the same in both the ideal and real worlds and the other requirements ensure that the inputs are the same, the only difference could be arbitration latencies. Thus we state the following requirement.

Requirement 15 (Conf Separation) *For all time t, the arbitration latency induced by every resource arbiter involved in granting execution resources for the evaluation of an interface call C, depends only on $\pi_{VM_i}(\sigma)$.*

7.3.4.4 Serialization of Interface Calls Finally, it has to be ensured that the sequence of interface calls by a VM depends on its own execution only. Operationally, the next instruction of the executing VM is pointed to by the program counter in the CPU, the updating of which depends on some combination of the address of the currently executing instruction and the execution of the instruction itself. Hence the integrity of this operation ensures that the next call depends on the execution context of the VM itself. Formally, we abstract this requirement as follows:

Requirement 16 (Next Call Separation) *For all virtual machines VM_i, time t, and any interface call C by VM_i, the next interface call of VM_i depends on $\pi_{VM_i}(\sigma)$ only.*

7.3.5 Our Condition for Isolation P_{iso}

We now define our requirement for isolation and summarize.

Definition 10 (P_{iso}) *XS We denote by P_{iso} the property that there exists a projection function π_{VM_i} for each VM VM_i, such that the conjunction of the Requirements 12-16 holds.*

Requirement 12 addresses explicit information flows, Requirements 13 and 14 cover storage side channels, and Requirement 15 covers timing side channels. Requirement 16 ensures

equivalence of control flow of VM executions in both the worlds. The next section establishes that these requirements yield provable isolation and in Sections 7.5 and 7.6 we perform further analysis of Requirements 15 and 12 respectively.

7.4 Proof of Sufficiency

This section develops a proof for isolation in a virtualized environment. We use the Universal Composability (UC) framework [Can] to model provable isolation as this framework allows us to assert security properties with the guarantee that they will be preserved when the system is arbitrarily composed with other systems. Further UC allows us to reason about the security of the interface *i.e.* isolation properties which will hold when principals (virtual machines), both benign and adversarial, use the platform interface. This is preferable to traditional approaches which focus on specific adversarial models. We start with an outline of the UC Framework and use that to specify a strong model of isolation and show that the requirements captured by P_{iso} are sufficient to achieve isolation in this model.

7.4.1 Isolation in the UC Framework

In the UC framework we can precisely define what we expect as isolation guarantees from a platform. The *ideal world* specification will be one in which each VM executes within its own platform and with its own copy of *all* resources. The *real world* is one in which multiple VMs execute within one platform and *share* a common set of resources. This is depicted in Figure 9.

The platform receives a set of VM images at the outset, one corresponding to each VM, and then just goes on executing them on its own. The only external interfaces are then the *Send* and *Receive* to the adversary A, which also communicates with $\mathcal{Z}$. A technical point here is that the platform interfaces to the VMs as outlined before is not available to the environment as is usual in the traditional UC models. So the environment is unaware of the details of the sequence of platform interface calls by the VMs. All it can do is to introduce the VM images. In the UC model, the platform has the following interfaces to the environment $\mathcal{Z}$, and the adversary A. (We omit the session ids [Can] here for ease of presentation. They can be easily added and would be needed for composition results):

IntroduceVM : In the real world, the platform receives a set of virtual machine images corresponding to VMs $VM_1, \cdots, VM_n$ from the environment $\mathcal{Z}$, and writes the images to the logical locations referred by the addresses in the VM's image. Some of these VMs may be *corrupt* and can send potentially useful information to the adversary A. Next, the platform sets the global timer to 0, and starts servicing the interface calls for all the VMs at time 0.

In the ideal world, there is a "demultiplex" step at the outset, where individual virtual machine images are sent to exclusive platforms. In this world, all the platforms set the global timer to 0 at the same time and for each VM_i, the corresponding platform starts servicing the interface calls at time 0.

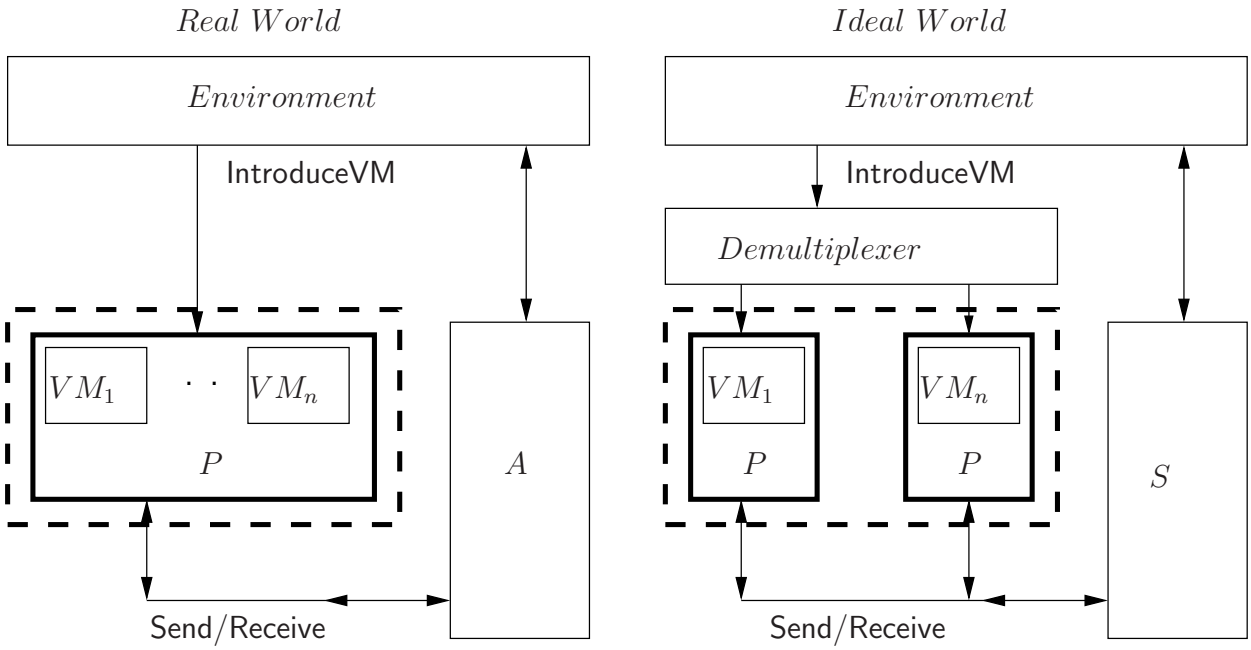


Figure 9: The two-world model for defining isolation in the UC framework. P denotes a platform consisting of a hypervisor and a contemporary computer.

Send : On a Send interface call from a VM, the message is sent to the adversary A.

Receive : On a Receive interface call from a VM, the message received from A for the VM, is returned to the VM.

We now define the ideal functionality $\mathcal{F}_P$, the execution of VMs on exclusive platforms which are naturally isolated by construction, and the real world implementation Π_P, the execution of VMs on a shared platform.

Definition 11 (Ideal Functionality) *The ideal functionality $\mathcal{F}_P$ is an execution where every VM gets an* exclusive *set of resources which are accessed through an* exclusive *platform.*

Definition 12 (Real-World Design) *The real-world design Π_P is an execution where all the VMs* share *a set of resources which are accessed through a* shared *platform.*

This section describes a proof where we show how the real-world platform provably realizes the idealized platform. First we use the UC framework to define when a platform achieves isolation.

Definition 13 (Isolation) *A platform P achieves isolation if the real-world design Π_P, with the given platform, securely realizes the ideal functionality $\mathcal{F}_P$, with the same platform.*

We show that the condition P_{iso} defined in Section 7.3.5 is sufficient to provably achieve isolation.

Theorem 2 *If a platform satisfies P_{iso} then it achieves isolation.*

Proof: We describe the outlines of a proof in the UC framework of the equivalence of the idealized platform with exclusive copies of resources with the real world platform providing access to shared resources. For this we have to show that the environment can not distinguish between the two models given any sequence of adversarial actions in the real world. That is we have to show that any attacks that are feasible in the real world can be simulated in the ideal world. Formally, we have to show that for any adversary A interacting with Π_P, there exists a *Simulator* S interacting with $\mathcal{F}_P$, such that the environment is not able to distinguish between the two interactions. Since we have abstracted all the external interactions through the Send and Receive interfaces we have to argue that the transcript of values in any sequence of Send and Receive will be the same in two cases.

At the outset, the environment uses the IntroduceVM call to send all the VM images corresponding to each of the VMs - in $\mathcal{F}_P$ they go to the exclusive systems and in Π_P they go to the shared system. After this note that the interfaces to the simulator S in the ideal world and the adversary A in the real world are exactly the same. Thus the simulator can simply replicate the actions of the adversary. In particular, when the simulator S receives a message $(\mathsf{Send}, VM_i, msg)$, it does exactly what the real world adversary does. Similarly it sends messages to individual platforms through the message $(\mathsf{Receive}, VM_i, msg)$ exactly when A does.

Given this simulation strategy to prove equivalence we have to prove by induction over time that the value of each *Send* will be the same in both worlds, at the *same time*, conditioned on the prior values of *Send* and *Receive* messages being the same. Observe that "corrupt" VMs may communicate useful information to A using the *Send* and *Receive* interfaces of the platform, but A does not have any additional interface to communicate with the platform or with any VM executing in the platform after the VM images of the VMs are introduced. Therefore, we have to prove that the transcript of communication to and from the adversary remains the same in both the ideal and real executions.

To prove that the network transcripts are equivalent, we observe from the interfaces that any message sent out by a VM depends on only the following: (i) values that it reads from its logical locations, (ii) results or errors returned by execution of operations, (iii) values received by it from the network, (iv) values returned by *ReadTimer* accesses. (v) sequence of the interface calls by the VMs. We prove by induction on time, that if the platform satisfies P_{iso}, then these values are identical for both worlds.

The base case of the induction is satisfied as follows: (i) The values at the logical locations for each VM are either from the VM image itself or initialized to $\perp$. (ii) No operation has been executed. (iii) There is no message received from the adversary yet. (iv) The VMs start at the same time in both worlds. (v) No operation has been executed. Observe that (i)-(v) are true in both the worlds at time 0.

Let the machine state in the real world be denoted σ^{real} and the individual machine states in the ideal world be denoted σ_i^{ideal} for each VM VM_i. The base case establishes that for all VMs VM_i, $\pi_{VM_i}(\sigma^{real}(0)) = \pi_{VM_i}(\sigma_i^{ideal}(0))$.

So now assume the induction hypothesis, that up to time t on the global clock in both the worlds, for all VMs VM_i, $\pi_{VM_i}(\sigma^{real}(t)) = \pi_{VM_i}(\sigma_i^{ideal}(t))$ and that all the items (i)-(v) are observed identically in both the worlds. We show that at time $t+1$ on the the global clock, for all VMs VM_i, $\pi_{VM_i}(\sigma^{real}(t+1)) = \pi_{VM_i}(\sigma_i^{ideal}(t+1))$ and that (i)-(v) are still identically observed at at $t+1$ on the global clock. Note that the global clock progresses in lock step in all the platforms in the ideal world.

Suppose the concurrent interface calls are $C_{l_1}, C_{l_2}, \cdots$ by the VMs $VM_{l_1}, VM_{l_2}, \cdots$ in the real world at time t. Consider any of these VMs VM_{l_k}. By Next Call Separation, the call C_{l_k} depends just on $\pi_{VM_{l_k}}(\sigma^{real}(t))$, which is equal to $\pi_{VM_{l_k}}(\sigma_{l_k}^{ideal}(t))$. By Conf Separation, the number of clock steps elapsed after the last call by VM_{l_k} just depends on $\pi_{VM_{l_k}}(\sigma^{real}(t'))$, where $t' < t$ is the time at the last interface call. By induction hypothesis, $\pi_{VM_{l_k}}(\sigma^{real}(t')) = \pi_{VM_{l_k}}(\sigma_{l_k}^{ideal}(t'))$. Hence the latency observed would be the same in both worlds.

Hence, in the ideal world, the same interface call comes up in the exclusive platform for VM_{l_k} at the same time. Then since P_{iso} is a sufficient condition for Observational Non-interference, we have (i) the results of the calls would be the same in both the worlds. (ii) the updated states for the VMs VM_{l_k} would satisfy: $\pi_{VM_{l_k}}(\sigma^{real}(t+1)) = \pi_{VM_{l_k}}(\sigma_{l_k}^{ideal}(t+1))$. (iii) for all other VMs VM_j, $\pi_{VM_j}(\sigma^{real}(t+1)) = \pi_{VM_j}(\sigma^{real}(t))$ and $\pi_{VM_j}(\sigma_j^{ideal}(t+1)) = \pi_{VM_j}(\sigma_j^{ideal}(t))$. Hence we will have for all VMs VM_i, $\pi_{VM_i}(\sigma^{real}(t+1)) = \pi_{VM_i}(\sigma_i^{ideal}(t+1))$. □

7.5 Formal Model of Conf Separation

In this section we consider the property of Conf Separation and investigate how one can achieve it in a real system. Towards this goal, we model the resource arbiters (i.e., schedulers, resource managers) in a hypervisor as deterministic finite-state systems, which form a suitable abstraction for many many popular classes of scheduling algorithms such as FIFO, priority-based, etc. Using our model we address the question of when and how one can achieve Conf Separation and more generally whether one can quantify the leakage in timing-based side channels.

7.5.1 Deterministic Finite-State Models of Resource Arbiters

In our model of resource arbiters, resource grants are made to VMs depending on the current state and the current input to the resource arbiter. This reflects the fact that in practice almost all resource arbiters are stateful where the state is reflective of all the pending requests and the particular policy. Our resource arbiters are similar to Mealy machines [Koh79], transitioning from state to state and producing outputs (i.e., resource grants) on the way. For simplicity, we do not model the details of the requests such as arguments, only whether there is a request.

Definition 14 (Deterministic Finite-State Resource Arbiter) *A deterministic finite-state resource arbiter (DFSRA) M is a tuple $(P, I, O, \Sigma, s_0, T)$ where:*

- *P is a finite set of VMs denoted $\{P_1, P_2, \cdots\}$,*
- *$I = 2^P$ is the input alphabet, where an input $i \in I$ denotes a set of concurrent requests from zero or more VMs,*
- *$O = 2^P$ is the output alphabet, where an output $o \in O$ denotes concurrent grants to zero or more VMs,*
- *Σ is a finite set of states $\{s_1, s_2, \cdots\}$,*
- *$s_0 \in \Sigma$ is a the start state, and*
- *T is the transition function $T : \Sigma \times I \rightarrow \Sigma \times O$, which takes a current state and a set of requests as input and produces a new state and a set of grants as output.*

States typically depend on past requests which may be queued, attributes about the queued requests and their corresponding VMs, and other such features. In this model an input trace is a sequence of concurrent requests (possibly empty) from VMs denoting step-wise temporal progress. Similarly, an output trace is a sequence of concurrent grants to VMs. For any request from an individual VM, the *arbitration latency* for the request is the number of steps between the request and the next grant to the same VM.

We define a execution trace generated by input trace $\mathcal{I} \in I^*$, and we write $exec_M(\mathcal{I}) \in O^*$, as the output trace obtained by running the DFSRA on the input trace $\mathcal{I}$, starting from the start state s_0, and following the states produced by the transition function T.

7.5.2 Common Resource-Arbitration Policies

We illustrate the expressibility of our DFSRA model by describing three common resource arbiters in our model, a priority-based scheduler, a first-in first-out (FIFO) scheduler, and a time-division multiplexing (TDM) scheduler. For each example, we assume that there are N requesters labeled P_0 through P_{N-1} which contend for a single resource and that the resource arbiter can make use of a queue of length l to track any resource requests that are not immediately granted.

7.5.2.1 Priority-based resource arbiter Our simple priority-based resource arbiter always satisfies the highest-priority request and queues ungranted requests. The requesters are completely ordered by priority, with P_0 having the highest priority and P_{N-1} the lowest. The states of the resource arbiter are the requesters in the request queue at any given point with the start state as the empty set $\emptyset$. If needed, we could additionally have an error state $\perp$.

The transitions are defined to reflect priority-based arbitration. Let the current set of requesters in the queue be Q and let P' be the set of requesters requesting at this time. The resource arbiter simply allocates the resource to the highest priority requester $\hat{P}$ in $Q \cup P'$ and queues all other requests. The resulting state after the transition is the state labeled

with $Q \cup P' - \{\hat{P}\}$. Variants could transition to error state if requesters already in the queue make requests.

7.5.2.2 First-in first-out (FIFO) resource arbiter Our FIFO resource arbiter can queue multiple requests from the same requester and further, it waits a minimum of one step before granting requests. The states of the FIFO DFSRA are the requesters in the request queue, of size l, and are viewed as ordered sequences to reflect the order of requests. The number of states is the number of possible sequences, that is, $1 + N + \cdots + N^l = \frac{N^{l+1}-1}{N-1}$.

If the current state reflects the sequence σ of requesters and the current input is from the subset P' of requesters, FIFO resource arbitration is simple to define. If the size of σ and the size of P' together is less than $l + 1$, we grant the resource to the requester at the head of σ (if not empty). The resulting state σ' dequeues this element and adds elements of P'. As before errors can be handled by transition to an error state.

7.5.2.3 Time-division multiplexing (TDM) resource arbiter The last example is a time-division multiplexing resource arbiter which assigns each requester a time slot in a periodic manner and grants requests to a requester exclusively in its slot. If the requester has no outstanding request during its time slot the TDM resource arbiter makes no grants. For each i, let requester P_i be alloted time slot i with periodicity N. States are labeled (r, Q), where r is the current slot number between 0 and $N - 1$ and Q the queued requests.

If the current state is labeled (r, Q) and P' is the set of requesters requesting at this time, the resource arbiter acts as follows: If P_r, the requester alloted this slot, is in $Q \cup P'$ then it is granted the resource else *no* requester is granted the resource. The resulting state is labeled $(r + 1 \ mod \ N, P' \cup Q - \{P_r\})$.

7.5.3 Conf Separation for Deterministic Finite-State Models

In this model, we can be assured of Conf Separation, or no timing side-channel leakage, only if the sequence of requests granted to any VM by the resource arbiter *only* depends on that VM's own input sequence. The rest of this section formalizes this intuition. We define the *projection* of resource arbitration algorithms to individual VMs *i.e.* the resource arbiter's behavior assuming that there are no requests from other VMs. Similarly, we can define the *product* of several individual resource arbiters as the single resource arbiter which jointly simulates the behavior of all the component arbiters if there are enough resources. The key result we are able to show is that Conf Separation is achieved only when all resource arbiters are *equivalent* in their input-output behavior to the product of their projection to individual VMs.

This section describes conditions under which resource arbiters can ensure that no information leaks due to arbitration decisions. From the perspective of a single VM, there is no interference or leakage of information from other VMs, if and only if whenever the VM makes identical sequences of requests the resource arbiter behaves exactly *identically*. This strong

requirement ensures that the VM is not able to infer anything about other VMs irrespective of what other VMs are running on the system thus achieving Conf Separation.

First we define *non-interference*, a necessary and sufficient condition for a DFS resource arbiter to achieve Conf Separation. Intuitively, this says that the projection of the execution trace from the DFSS for a particular VM depends only on the projection of the input trace to the requests of the particular VM. To formalize this intuition we need the following definitions.

Definition 15 (Projection of a Trace) *Given an input trace $\mathcal{I}(= \langle i_1, i_2, \cdots \rangle)$, the projection $\pi_{P_i}(\mathcal{I})$ with respect to a VM P_i is the sequence $\langle i'_1, i'_2, \cdots \rangle$ where $i'_k = i_k \cap \{P_i\}$. Similarly we can project output traces.*

A resource arbiter is non-interfering with respect to VM P_i, if whenever P_i makes an identical sequence of requests, the resource arbiter produces an identical sequence of request grants/denies for P_i. Note that two sequences are identical if they contain the same requests (or grants) in the same relative order and with the same inter-arrival time.

Definition 16 (Non-interference) *A resource arbiter M is non-interfering with respect to a VM P_i if the following property holds true:*

$$\forall \mathcal{I}_1, \mathcal{I}_2 \in I^* \; . \; \pi_{P_i}(\mathcal{I}_1) = \pi_{P_i}(\mathcal{I}_2) \implies \pi_{P_i}(exec_M(\mathcal{I}_1)) = \pi_{P_i}(exec_M(\mathcal{I}_2)) \quad (1)$$

A resource arbiter is non-interfering if it is non-interfering with respect to all VMs.

Consider a resource arbiter which is non-interfering with respect to two VMs P_1 and P_2 contending for a single resource, where both VMs make a request at some time instant k. Consider the resource arbiter output on the projections $\pi_{P_1}(\mathcal{I})$. Since the resource arbiter is non-interfering with respect to P_1 if it grants to P_1 at time k on $\pi_{p_1}(\mathcal{I})$ then it must do so on input $\mathcal{I}$. This is also simultaneously true for VM P_2. Thus a non-interfering resource arbiter must make the same grants at the same time irrespective of which VMs make requests. For such resource arbiters, the outputs at each step can simply be obtained by looking at the outputs made on projections. In the rest of the section we show formally that a non-interfering resource arbiter is equivalent to the one obtained by composing its per-VM projections.

We extend the definition of projection with respect to a VM from a trace to a DFS resource arbiter by restricting the resource arbiter to only consider inputs from that VM.

Definition 17 (Projection of a Resource Arbiter) *The projection of a resource arbiter $M = (P, I, O, \Sigma, s_0, T)$ on a VM P_i is resource arbiter $M' = \pi_{P_i}(M) = (P', I', O', \Sigma, s'_0, T')$ where:*

- *The set of VMs P' is just $\{P_i\}$, set of inputs I' and outputs O' are each just $2^{P'}$.*

- *The transition function T' is constructed from T by removing all transitions $(s_1, i) \rightarrow (s_2, o)$, where i contains requests other than from P_i.*

After removing unreachable states, if there is a remaining transition with a grant to a VM other than P_i then the projection is not well-defined, since then there is a grant to a VM who never makes a request.

The product of two resource arbiters is one that jointly simulates the behaviors of the component resource arbiters when possible.

Definition 18 (Product) *Given two resource arbiters $M_1 = (P_1, I_1, O_1, \Sigma_1, s_0, T_1)$ and $M_2 = (P_2, I_2, O_2, \Sigma_2, t_0, T_2)$, with* disjoint *sets of VMs, we define their product $M_1 \times M_2 = (P, I, O, \Sigma, r_0, T)$, as follows: The set of VMs P is the union of the set of VMs $P_1 \cup P_2$ The input set I and the output set O are both the power set of P. The set of states Σ is the product of the two sets of states $\Sigma_1 \times \Sigma_2$. For all transitions $(s_1, i) \rightarrow (s_2, o)$ in T_1 and all transitions $(t_1, i') \rightarrow (t_2, o')$ in T_2, we introduce the following transition in T: $((s_1, t_1), i \cup i') \rightarrow ((s_2, t_2), o \cup o')$.*

We define the equivalence of two resource arbiters purely in terms of their input-output behavior. In particular, it requires nothing about the internal structure of the states and transitions themselves. Two equivalent resource arbiters may have completely different internal policy, states and transitions.

Definition 19 (Equivalence) *Two resource arbiters M_1 and M_2 are equivalent, written $M_1 \equiv M_2$, if for any input trace $\mathcal{I}$, $exec_{M_1}(\mathcal{I}) = exec_{M_2}(\mathcal{I})$.*

The following is the central result of this section and captures precisely when resource arbiters are non-interfering.

Theorem 3 *A resource arbiter $M = (P, I, O, \Sigma, s_0, T)$ for VMs $P = \{P_1, \cdots, P_k\}$ is non-interfering with respect to all the VMs in P if and only if $M \equiv \pi_{P_1}(M) \times \cdots \times \pi_{P_k}(M)$ where all the projections have to be well-defined. Further the above equivalence can be checked algorithmically.*

Proof: Let $M' = \pi_{P_1}(M) \times \cdots \times \pi_{P_k}(M)$. Since a product resource arbiter is, by definition, non-interfering with respect to each of the VMs, If M is equivalent to M' then it is non-interfering. To see the other direction, we first note that for the projection trace $\mathcal{I}_i = \pi_{P_i}(\mathcal{I})$, we have $exec_M(\mathcal{I}_i) = exec_{\pi_{P_i}(M)}(\mathcal{I}_i)$ since the projection trace has requests only from P_i. Let the composition '$|$' of two traces $\langle i_1, i_2, \cdots \rangle$ and $\langle j_1, j_2, \cdots \rangle$ be the sequence $\langle i_1 \cup j_1, i_2 \cup j_2, \cdots \rangle$. Note that, by definition, M' outputs $exec_{\pi_{P_1}(M)}(\mathcal{I}_1) \mid \cdots \mid exec_{\pi_{P_k}(M)}(\mathcal{I}_k)$ on input $\mathcal{I}_1 \mid \cdots \mid \mathcal{I}_k$. If M is non-interfering we have $\pi_{P_i}(exec_M(\mathcal{I})) = \pi_{P_i}(exec_M(\mathcal{I}_i))$ because $\mathcal{I}$ and $\mathcal{I}_i$ agree on the inputs for P_i. Therefore $\pi_{P_i}(exec_M(\mathcal{I})) = \pi_{P_i}(exec_{\pi_{P_i}(M)}(\mathcal{I}_i)) =$

$exec_{\pi_{P_i}(M)}(\mathcal{I}_i)$. Since this is true for all VMs P_i, we have

$$\begin{aligned}
\therefore \pi_{P_1}(exec_M(\mathcal{I})) \mid \cdots \mid \pi_{P_k}(exec_M(\mathcal{I})) \\
&= exec_{\pi_{P_1}(M)}(\mathcal{I}_1) \mid \cdots \mid exec_{\pi_{P_k}(M)}(\mathcal{I}_k) \\
&= exec_{M'}(\mathcal{I}_1 \mid \cdots \mid \mathcal{I}_k) \\
\therefore exec_M(\mathcal{I}) &= exec_{M'}(\mathcal{I})
\end{aligned}$$

As the above holds for any input sequence $\mathcal{I}$, $M \equiv M'$.

Since a resource arbiter as defined in Definition 14 is a Mealy machine [Koh79] over input alphabet I, output alphabet O, set of states Σ, state transition function T, and start state s_0, there is a simple algorithm to check if two resource arbiters are equivalent based on the standard minimization procedure for Mealy machines [Koh79] which produces a minimal, equivalent resource arbiter. To determine equivalence, two resource arbiters M_1 and M_2 are reduced to their minimal equivalent resource arbiters and then we can determine if there is an isomorphism between the two minimal resource arbiters. □

7.5.4 Non-Interfering Resource Arbiters

Consider a non-interfering DFS resource arbiter M which handles requests from two VMs P_1 and P_2 contending for a single resource. By Theorem 3, M is equivalent to $M_1 \times M_2$ which are the projections to P_1 and P_2 respectively. We show that if M is not a time-division multiplexing (TDM) DFS resource arbiter then there will be a state in which M must make grants to both VMs which violates availability.

States in M are of the form $u = (s, t)$ where s and t are states in M_1 and M_2. u is reachable in M if and only if there are paths of length k to reach s in M_1 and t in M_2. Since there is exactly one resource, there can be no reachable state $(\hat{s}, \hat{t})$ in M where $\hat{s}$ grants P_1 in M_1 and $\hat{t}$ grants P_2 in M_2 on any outgoing edge. Thus the set of all possible path lengths from s_0 to $\hat{s}$ and the set of all possible path lengths from t_0 to $\hat{t}$ are disjoint.

The set of all possible path lengths from s_0 to $\hat{s}$ is the union of a finite set and a periodic set with period equal to the least common multiple (lcm) of all possible simple cycle lengths in the subgraph induced by all the nodes in any path between s_0 and $\hat{s}$. Thus if L is the lcm then if there is a path of length k to $\hat{s}$ then there is also a path of length $k + L$. This is also similarly true for $\hat{t}$ in M_2. If the periods are L_1 and L_2 respectively, then consider the sets as periodic sets of period $L = lcm(L_1, L_2)$. Then since M can allocate only one resource at a time, in any period of length L, VMs P_1 and P_2 can be granted only at pre-determined fixed slots. Thus M is a TDM DFS resource arbiter!

For m VMs and n resources, where $m > n$, the constraint is that there is no integer $k \geq 0$, such that there are more than n of the projection resource arbiters with a path of length k from the start state to a resource granting state. Observe that a strong condition, as for the case of two VMs and one resource, is not necessary here. For example, for five VMs and three resources, two of the VMs can be freely assigned to two resources (one per resource), while TDM-arbitrating the remaining three on the third resource. We conjecture in this case that at least $m - n + 1$ VMs *have* to be TDM arbitrated.

7.5.5 Probabilistic Models of Resource Arbiters

Given that the only deterministic resource arbiters which achieve Conf Separation are inefficient, we extend the model to allow for resource arbiters which take decisions probabilistically and define a leakage notion associated with executing multiple VMs in this model. This would allow one to trade off efficiency for leakage by comparing the added performance of moving from TDM resource arbitration to probabilistic resource arbitration against the leakage rate bounds guaranteed by the probabilistic model.

Definition 20 (Probabilistic Resource Arbiter) *A* probabilistic resource arbiter (PRA) *is a randomized algorithm where in a given state the algorithm can probabilistically grant the requests of a subset of VMs on a given input.*

Note that the execution trace of a PRA is now a *distribution*, rather than a single output trace.

Consider two VMs A and P and a probabilistic resource arbiter M. For a given sequence length k, let I_P be the random variable denoting the input sequence from P. For a *fixed* individual input sequence i_A from A, the output of the resource arbiter projected to A's grants is a distribution fixed by the distribution of P's requests and the random bits used by M. We denote this distribution by $O_A(i_A)$.

Definition 21 (Leakage Rate) *The leakage rate from a VM P to a VM A in a probabilistic resource arbiter M is the rate of difference in entropy between the a priori distribution of I_P and the conditional entropy of I_P given $O_A(i_A)$, maximized over all possible i_A's.*

If $S_k(A)$ is the set of all A's input sequences of length k, the leakage rate from P to A is:

$$Leakage\ Rate = \lim_{k\to\infty}\left[\max_{i_A\in S_k(A)}\left(\frac{H(I_P) - H(I_P \mid O_A(i_A))}{k}\right)\right].$$

Our definition assumes that A is non-adaptive in the sense that its request sequence is fixed regardless of the grants it observes from the resource arbiter.

7.5.5.1 Example of leakage-rate computation We give an example of a probabilistic resource arbiter, for which we compute the leakage rate. Our example is a stateless resource arbiter with two VMs $\{P_A, P_B\}$ that makes a probabilistic decision at each step. It does not queue requests and hence does not grant to processes which do not make requests at the current time. If only P_A makes a request the resource arbiter grants with probability p, if only P_B requests it grants with probability q. If both request then P_A is granted with probability r, P_B with probability s and neither with probability $1-(r+s)$. Since the resource arbiter is stateless and we assume that the input trace is non-adaptive, the leakage rate is equal to the leakage of a single step. Assume that the a priori distribution of requests from P_B is uniform. Observe that, if there is no request by P_A, $H(I_{P_B}|O_{P_A})$ still remains equal to $H(I_{P_B}) = 1$, since there is no grant to P_A.

When P_A makes a request, based on whether it was granted, P_A can infer information about the distribution of P_B's request. Let $I_B = 0$ ($I_B = 1$) be the events that P_B does not request (requests) and $O_A = 0$ ($O_A = 1$) be the events that P_A does not get (gets) a grant. It is straightforward to show that $Pr[O_A = 0] = (2 - p - r)/2$ when P_A makes a request. Using this we can derive

$$\begin{aligned} Pr[I_B = 0|O_A = 0] &= \frac{Pr[O_A = 0|I_B = 0]\ Pr[I_B = 0]}{Pr[O_A = 0]} \\ &= \frac{1-p}{2-p-r}. \end{aligned}$$

Similarly, one can compute $Pr[I_B = 1|O_A = 0]$ to be $(1 - r)/(2 - p - r)$ using which we can precisely compute $H(I_B|O_A = 0)$. We can similarly compute $H(I_B|O_A = 1)$. Finally we have

$$\begin{aligned} &H(I_B|O_A) \\ &\quad = Pr[O_A = 0]H(I_B|O_A = 0) + Pr[O_A = 1]H(I_B|O_A = 1) \\ &\quad = \left(1 - \frac{p+r}{2}\right) H(I_B|O_A = 0) + \frac{p+r}{2} H(I_B|O_A = 1), \end{aligned}$$

and thus the leakage rate is $1 - H(I_B|O_A)$.

7.6 Formal Model of Loc Separation

Loc Separation deals with reads and writes of memory locations by VMs. In this section, we derive lower-level requirements for Loc Separation as requirements for the memory access control mechanisms employed by a platform consisting of hardware and a type I or type II hypervisor as well as requirements for the context switch mechanism employed by the hypervisor. We derive these requirements in two steps. First, we derive the requirements for access control assuming a *static partitioning* of the platform memory locations among VMs. Second, we derive the requirements for the context switch mechanism when the partitions are dynamically modified by the hypervisor.

7.6.1 Static Partitions

The memory locations used by VMs are mapped to physical memory locations on the platform. This mapping takes the form of a translation map, which is an identity mapping for certain physical memory locations. A hypervisor partitions the physical memory locations of the platform among VM. Processing elements then perform reads and writes of physical memory locations when requested by VMs via *ReadLoc* and *WriteLoc* interface calls. The processing elements of the platform can be partitioned into two sets. Borrowing terminology from computer systems, we call these partitions CPU and peripheral devices. In the CPU partition we include hardware CPU as well as software-defined PE such as virtual devices,

hypercalls, and system calls. The peripheral devices partition includes all hardware devices but the CPU. We begin with some definitions.

Definition 22 (Memory Elements) *There are four types of memory elements in the platform: (1) registers of all CPUs C, (2) main memory words M, (3) registers of all peripheral devices P, and (4) persistent storage locations S, such as hard drive disk sectors. Elements in the set P are of two types: memory-mapped elements and IO ports. Each memory element has a unique label called* physical address*. Processing elements read and write memory elements using interface calls $ReadME(physicaladdress)$ and $WriteME(physicaladdress, val)$ respectively.*

We state the following axiom on the operational model of memory elements.

Axiom 1 (Persistence of memory elements) *For all physical addresses $addr$, the value returned by $ReadME(addr)$ is val, if the immediately last write to the same address $addr$ was val, that is the immediately last write interface call to $addr$ was $WriteME(addr, val)$. If there was no previous write to $addr$, then $ReadME(addr)$ returns the constant bit string $\perp$.*

Definition 23 (Namespace) *Namespace is a set of identifiers used by VMs to access memory locations. It is the set of tuples $(VM, location)$.*

Definition 24 (Translation Map) *A Translation Map is a function mapping from a namespace element to a physical address. In actual systems, only memory elements in M and memory-mapped memory elements in P have non-identity translation maps.*

7.6.1.1 Operational Model of Processing Elements We now present an operational model of processing elements that describes the interfaces via which they access memory elements.

Operational Model of CPU A CPU can read and write any memory element in the sets M, P, and all of its registers. A CPU cannot read or write the registers of other CPUs.

On getting an interface call $ReadLoc(VM_i, loc)$, where the namespace element (VM_i, loc) is mapped to a memory element of the set M or to a memory-mapped element of the set P, the CPU returns the value returned by $ReadME(TranslationMap(VM_i, loc))$. The translation map is set up by the hypervisor and the function is computed by a hardware device such as the memory management unit (MMU). Similarly for writes, we have $WriteME(TranslationMap(VM_i, loc), val)$. If no translation exists for a namespace element, the MMU notifies the condition by raising an interrupt. The interrupt is handled by the hypervisor, which could add memory elements to resource partition of VM_i and update the translation table to map (VM_i, loc) to the newly allocated resource, or it could signal an error to VM_i.

Access from CPU to elements in P which are IO ports and elements in S is governed by a suitable access control policy; examples of mechanisms used to implement such access control policies include the IO Permission Bitmap of x86 CPU and file system access control. On getting an interface call $ReadLoc(VM_i, loc)$, where the namespace element (VM_i, loc) is mapped to read a memory element of the set P that is an IO port, or to a memory element in the set S, the CPU returns the value returned by $ReadME(AccessControlPolicy(VM_i, loc))$. Similarly we have $WriteME(AccessControlPolicy(VM_i, loc), val)$, for writes to IO ports and memory elements in the set S. If VM_i cannot read or write to port or storage location at (VM_i, loc) due to the access control policy, the CPU notifies of the condition by raising an interrupt. The interrupt is handled by the hypervisor, which could update the access permissions to allow VM_i to access loc, or it could signal an error to VM_i.

Operational Model of Peripheral Devices A peripheral device can read and write any memory element in the set M via hardware mechanisms such as direct memory access (DMA) and memory elements in the set P. A peripheral device cannot read any memory elements in the set C or S. However, it can read and write the memory elements in S indirectly via reads and writes of memory elements in the set P belonging to the disk controller.

On receiving an interface call $ReadLoc(VM_i, loc)$ where (VM_i, loc) is mapped to a memory element of the set M, a peripheral device returns $ReadME(TranslationMap(VM_i, loc))$. The translation map is set up by the hypervisor and the function is computed by a hardware device such as the IO memory management unit (IOMMU). Similarly, for writes we have $WriteME(TranslationMap(VM_i, loc), val)$. If no translation exists for a namespace element, the IOMMU notifies the condition by raising an interrupt. The interrupt is handled by the hypervisor, which could add memory elements to resource partition of VM_i and update the translation table to map (VM_i, loc) to the newly allocated resource, or it could signal an error to the peripheral device.

Access from a peripheral device to all elements in P is governed by a suitable access control policy. Such access control policies can be implemented using hardware mechanisms such as PCI Express Access Control Services [PCI06]. On getting an interface call $ReadLoc(VM_i, loc)$ to read a memory element of the set P, the peripheral device returns the value returned by $ReadME(AccessControlPolicy(VM_i, loc))$. Similarly for write we have $WriteME(AccessControlPolicy(VM_i, loc), val)$. If VM_i cannot read or write the memory element of the set P at loc, the peripheral device notifies the condition by raising an interrupt. The interrupt is handled by the hypervisor, which could update the access permissions to allow VM_i to access loc, or it could signal an error to the peripheral device.

7.6.1.2 Requirements for Static Partitions

The requirement can be stated as follows:

Requirement 17 *(i) The translation map is an injective function, possibly partial, and (ii) the access control policy must only permit a VM to read and write memory elements to which it has exclusive access.*

The basic soundness proof idea is to bootstrap from the axiom about the persistence of MEs and use injectivity of the translation map function to prove the existence of Loc Separation. The translation map can be partial to account for the fact that all namespace elements may not have a translation. For memory elements that have an identity translation map, the access control policy must ensure that any VM having access to these memory elements has an exclusive access. This ensures that there is no explicit flow of information through shared memory.

7.6.2 Modification of Partitions

A hypervisor modifies the partitions of the memory elements either as a result of an interrupt from a CPU or peripheral device indicating that a namespace element has no translation, or as a result of an explicit allocation request from a VM. Therefore, the partitions change over time. The following requirement must be asserted at the completion of each such modification of partitions.

Requirement 18 (Value Invariance) *Suppose that the partition for VM changes from P_1 to P_2. Then, its translation map changes from T_1 to T_2. Consider each namespace element (VM, loc), such that (VM, loc) maps to different physical addresses under mappings T_1 and T_2. That is $T_1(VM, loc)$ is a different physical address than $T_2(VM, loc)$. Then it is required that any subsequent read call $ReadME(T_2(VM, loc))$ returns val, where val was the immediately last value written to $T_1(VM, loc)$ using $WriteME(T_1(VM, loc), val)$, before the partition was modified.*

7.7 Discussion

With this work we have taken a different approach towards system development. In the conventional approach, one develops a high-level specification that defends against known attacks, and formally proves that its low-level implementation satisfies the specification. We contrast this with our approach in the UC framework (Section 7.4), where an ideal-world model captures the security properties that we want to achieve, and the interfaces that are exposed to attack (the security assumptions). We then prove that these security properties hold for *all possible attacks* on those interfaces in our real-world model.

Our UC-based approach differs in the granularity of attacks addressed. While the conventional approach focuses on specific symptoms (the attacks), our work focuses on underlying causes (the mechanisms that make such attacks possible). Our Conf Separation condition (Section 7.5) addresses the root cause of timing side channels: competition for shared resources whether these are CPU cycles, cache lines, or access to a disk drive. We prove that timing side channels due to resource arbitration latencies can be eliminated with static time partitioning. Our Loc Separation condition (Section 7.6) addresses the root cause of explicit information flows. We demonstrate that access control (via address translation or access matrix mechanisms) and correct execution of context switch operations are sufficient to prevent explicit information flows.

Our isolation conditions are at least in part realizable in hardware, as demonstrated by two secure processor implementations that eliminate implicit information flows. The first, GLIFT [TWM+09], uses dynamic information flow tracking with shadow circuitry to make all information flows explicit. The second, Caisson [LTO+11], provides provable information flow security of micro-architectural processor features, using a statically-verifiable hardware design language.

Besides addressing attacks at higher levels of granularity, the UC framework encourages modular design. In particular, UC guarantees that security properties of one system will hold even when that system is composed with other systems, without introducing security vulnerabilities due to composition. Just like modular design of systems leads to economies of scale with pre-designed components, we believe that the UC framework can lead to economies of scale with modular proofs constructed by composition.

7.8 Summary

Virtualization platforms are built around the concept of transparent sharing of hardware resources and have increasingly been used to provide isolation between the virtual machines of different users or at different security levels. Unfortunately, such isolation is incomplete as many attacks have demonstrated.

We presented a formal model for isolation that covers both explicit information flows and timing-based and storage-based implicit information flows. To make this formal model practically applicable, we presented five requirements for isolation, each one addressed to a different component of a hypervisor and hardware platform (access control, error reporting, implicit state, arbitration policies, and execution control). We proved in the Universal Composability framework that for isolation to be realized in a practical virtualization platform it is sufficient that the five requirements are satisfied.

In our formal treatment of timing side channels we provided an isolation proof for deterministic resource arbiters and showed that only time-division multiplexing (TDM) scheduling policies can achieve isolation. Since TDM schedulers are not favored due to their inefficiency, we considered a weaker definition of isolation that allows a bounded amount of leakage and quantified leakage rates for a subclass of probabilistic stateless schedulers.

In our formal treatment of explicit information flows we demonstrated that access control (via address translation or access matrix mechanisms) and context switching are sufficient to prevent such flows. Our results cover a wide spectrum of virtualization platforms, from paravirtualization to dynamic binary translation to self-virtualizing hardware.

We believe that our formal discussion of isolation will guide software implementors to take advantage of features that enable sharing without explicit or implicit information flows, and that it will guide hardware implementors to provide features that support efficient resource sharing without information leakage.

This application of the UC Framework to model a complex systems concept such as strong isolation is evidence of the general suitability of the framework to other applications.

8 Composition Failures in Protocols

The Universal Composability framework was originally developed for and has been extensively used to model cryptographic protocols (see [Can, Can06]). Its formal mechanisms are well suited to the message passing paradigm where the interfaces can be essentially which messages the parties actually respond to. In the Montage project, we have tried to apply UC to protocols outside the realm of cryptography in order to capture protocols which have elements of system implementations. Further as we discuss in Section 10 they provide nice examples for our effort to automate the equivalence proofs. In the course of our investigation of the right protocols to model, we discovered a very general protocol composition failure in a number of practical systems and daemons. This is a perfect example of the kinds of problems which can be avoided by modeling with a framework which is composable.

8.1 Plaintext injection in multiple legacy protocol implementations

This segment details the software flaw that was initially discovered by Wietse Venema, one of the Montage team members, while maintaining the Postfix open source mail server [Ven] implementation of SMTP (Simple Mail Transfer Protocol) [Kle08] over TLS (Transport Layer Security) [DA99].

Subsequent investigation revealed that this flaw was widespread. It affected not only multiple implementations of SMTP over TLS, but it also affected multiple implementations of *other* Internet protocols over TLS. An overview of known to be affected products and services is shown in Table 1. We present an overview of the problem and its impact, and draw lessons about where one can expect to find similar problems.

8.1.1 Problem overview and impact

The TLS protocol [DA99] encrypts communication and protects it against modification in transit across the network. This protection exists only if a) software is free of flaws, and b) clients verify the server's TLS certificate, so that there can be no "man in the middle" (servers usually don't verify client certificates).

The problem discussed here is caused by a software flaw that affected the TLS support for multiple legacy Internet protocols including SMTP [Kle08], IMAP (Internet Message Access Protocol) [Cri03], POP (Post Office Protocol) [MR96], FTP (File Transfer Protocol) [PR85] and NNTP (Network News Transfer Protocol) [Fea06].

In the case of SMTP, the flaw would allow an attacker to inject client commands into an SMTP session during the unprotected plaintext SMTP protocol phase (described in more detail later), such that the server would execute those commands during the SMTP-over-TLS protocol phase (when all communication is supposed to be protected). The injected commands could, for example, be used to steal the victim's email or SASL (Simple Authentication and Security Layer) [SM07] username and password.

Table 1: Summary of known to be affected products and services, with vulnerability identifiers if available. The proprietary applications used in-house for the Gmail and Postini services were fixed without a public announcement.

Vulnerability ID	Affected Product or Service
CVE-2011-0411	Postfix mail server [Ven11]
CVE-2011-0411	Oracle Communications Messaging Server [Ora11a]
CVE-2011-1430	Ipswitch IMail [Ima11]
CVE-2011-1431	qmail-tls [qma11] (third-party patch for qmail [net] mail server)
CVE-2011-1432	SCO SCOoffice Server
CVE-2011-1506	Kerio Mailserver [ker11]
CVE-2011-1575	Pure-FTPd [Den11]
CVE-2011-1926	Cyrus IMAP [cyr11] (IMAP, LMTP, POP3, NNTP server)
CVE-2011-2165	WatchGuard XCS [wat11]
CVE-2012-0070	spamdyke [spa12] (third-party front-end for qmail [net] mail server)
N/A	qpsmtpd [qps11] (third-party front-end for qmail [net] mail server)
N/A	Gmail mail submission service
N/A	Postini mail filtering service

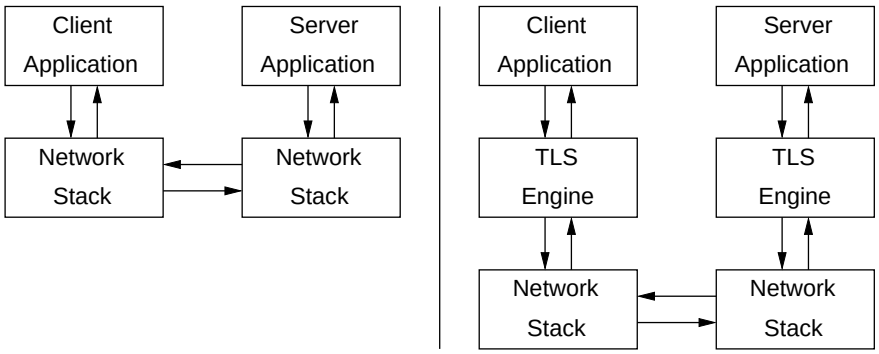


Figure 10: Logical view of client-server application before (left) and after (right) the START-TLS client request.

8.1.2 The STARTTLS feature

STARTTLS is the generic name for a feature of many legacy Internet protocols. Its purpose is to "upgrade" a session from plaintext to TLS.Although web servers use different TCP ports for plaintext (`http` on port 80) and encrypted (`https` on port 443) communication, the use of separate ports for TLS and non-TLS communication has been deprecated for other protocols. Some arguments for depreciation can be found in Section 7 of [New99]. STARTTLS implementations exist among others for SMTP [Hof02], IMAP [New99], POP [New99], FTP [FH05] and NNTP [MVN06].

The basic idea of STARTTLS is that a client connects to the standard network port for the legacy service (for example TCP port 25 for SMTP or 143 for IMAP), and then performs the traditional plaintext handshake for that protocol. The plaintext handshake is needed because that is how the protocol is defined; simply skipping the plaintext phase would break interoperability.

During the plaintext handshake a server may announce that it supports TLS. If both client and server support TLS, the client may send a STARTTLS request to turn on TLS (in some protocols the command is abbreviated to STLS). Logically, this request results in the insertion of a TLS engine between the application and the network stack as illustrated in Figure 10.

After the client- and server-side TLS engines negotiate a TLS cipher and session key, all further traffic between the client and server application is encrypted and protected against modification in transit across the network. The client and server application discard all information that they exchanged during the plaintext handshake, and restart their conversation using the same application protocol command set as they used in the absence of TLS encryption.

8.1.3 Demonstration of the problem for SMTP

The problem is easy to demonstrate with a one-line change to the OpenSSL `s_client` command source code. This program makes a connection to servers that support straight TLS, SMTP over TLS, or a handful other protocols over TLS. The demonstration here focuses on SMTP over TLS only.

The demonstration with SMTP over TLS involves a one-line change in the OpenSSL s_client source code, available from `openssl.org`. With OpenSSL 1.0.0, the change is made at line 1129 of file `apps/s_client.c`. To build the `s_client` command, execute the commands `./config; make`.

- Old code: `BIO_printf(sbio,"STARTTLS\r\n");`
- New code: `BIO_printf(sbio,"STARTTLS\r\nRSET\r\n");`

The modified `s_client` command sends the plaintext `STARTTLS` command immediately followed by an RSET command (a relatively harmless SMTP protocol "reset)". Both commands are sent as plaintext in the same TCP/IP packet, and arrive together at the server.

The \r\n are the carriage-return and newline characters; these are necessary to terminate an SMTP command.

When an SMTP server has the plaintext injection flaw, it reads the STARTTLS command first, switches to SMTP-over-TLS mode, and only then the server reads the RSET command. Note, the RSET command was transmitted during the plaintext SMTP phase when there was no protection, but the server reads the command as if it was received over the TLS-protected channel.

Thus, when the SMTP server has the flaw, the s_client command output will show two 250 SMTP server responses instead of one. The first 250 response is normal, and is present even when the server is not flawed. The second 250 response is for the RSET command, and indicates that the SMTP server has the plaintext injection flaw.

```
$ apps/openssl s_client -quiet -starttls smtp -connect server:port
[some server TLS certificate details omitted]
250 some text here [Normal response, also with "good" server]
250 more text here [RSET response, only with flawed server]
```

An attacker would exploit this by playing man-in-the-middle on the connection between SMTP client and server, perhaps using one of many available tools for ARP [Plu82] spoofing at a public WIFI access point. Instead of adding a harmless RSET command, the attacker could inject commands into a victim's session to open a mail transaction, and steal the victim's SASL authentication credentials and email.

8.1.4 Switching world views

When the switch from plaintext to TLS mode is made, all layers above the TLS engine (see Figure 10) need to purge all information that they received through the plaintext handshake. This world view switch needs to be implemented consistently. Otherwise, "extra" information that was sent as unprotected plaintext may slip through the cracks.

In the case of Postfix and other affected implementations, this world view switch was incomplete. These implementations did not account for information that lingered in stream buffers at the boundary between software layers. This allowed an attacker to append "extra" commands to the plaintext STARTTLS request, such that those commands would be read after the switch to TLS was completed, as if the commands had arrived through the TLS-encrypted session.

8.2 Remediation

There are two types of remedies to address the flaw: those that focus on the buffering problem, and those that focus on unexpected command or response pipelining. Both remedies can be used together.

- Clear buffers: After switching from plaintext to TLS, client and server applications must discard all contents of their network input buffers, just like they must discard all SMTP protocol information received during the plaintext handshake.

- Server-side pipelining check: Report an error when a server receives unexpected input appended to a STARTTLS command. For example, RFC 3207 [Hof02] states that STARTTLS must be the last SMTP command in a pipelined group. Similar constraints are specified for other protocol-over-TLS implementations. If a plaintext command is appended to STARTTLS, then that is a protocol violation.

  This remedy can also be implemented outside the mail server, for example in a protocol-aware firewall or other network-level tool.

- Client-side pipelining check: Report an error when the client receives server responses after TLS is turned on, for commands that the client did not yet send.

  This measure is complementary to the server-side pipelining check. Since the unexpected server replies are sent through the TLS-protected session, this remedy cannot be implemented in network-level tools.

8.3 Comparison with other vulnerabilities

In 2009, two independent groups jointly announced a new vulnerability in x86 processors [Duf09, WR09]. This vulnerability involved SMM RAM, a region of main memory in x86 CPUs that is accessible only when the CPU executes an SMM (System Management Mode) interrupt. In SMM mode, the CPU runs at a privilege level that is higher than root (supervisor) or hypervisor. If an attacker could compromise SMM RAM, they would be undetectable by software running with less-than-SMM privileges.

In the attack, a program with root or hypervisor privilege configures a memory controller such that the SMM RAM address range was changed from non-cacheable into "cacheable with write-back", then writes machine instructions to SMM RAM memory. The CPU writes the instructions to the CPU cache; this is possible because the CPU cache doesn't know whether the CPU can write to SMM RAM. Once the CPU is triggered to switch into the privileged SMM mode, it executes the attacker's SMM RAM instructions from the cache.

This CPU cache poisoning vulnerability is similar to STARTTLS plaintext injection. In both cases, there are a lower- and higher-security context; instructions are stored in a lower-security context, and are executed after switching to a higher-security context. Both cases, the solution is to discard such stored instructions when switching to the higher-security context.

The STARTTLS plaintext injection problem differs from cache poisoning attacks that are possible with unauthenticated implementations of DNS (domain name system) [Moc87], ARP (address resolution protocol) [Plu82] or DHCP (dynamic host configuration protocol) [Dro97], where an attacker attempts to send forged answers before the real reply arrives.

With these attacks, the problem is that the victim cannot reliably distinguish between an authentic response and a forgery; all the attacker needs to do is win a race.

8.4 Summary

The plaintext injection problem described in this segment happens when two conditions are met:

- A protocol has a STARTTLS-equivalent feature that allows two communicating applications to switch from plaintext to encrypted communication, by inserting a cryptographic protocol underneath the plaintext application protocol (see Figure 10).

  This is a popular approach to "upgrade" legacy Internet protocols with the security benefits from TLS, without requiring any changes to the legacy protocols.

- A server implementation supports command pipelining, such that plaintext commands, appended to the STARTTLS-equivalent command, will be executed after the plaintext-to-ciphertext switch is completed.

  As many legacy Internet protocol implementations use some form of I/O buffering internally, they will unwittingly support command pipelining even when this is not a documented protocol option.

In conclusion, plaintext injection will be a recurring problem for as long as applications have a STARTTLS-equivalent feature.

9 Modeling of OAuth 2.0 Web Security Protocol

Web security protocols are very natural candidates for analysis with the UC framework. Typically, they are primarily cryptographic protocols with a bit of infrastructure elements like DNS and systems such as browser implementations and conventions. Besides analyzing important real-world web security protocols thereby ensuring that they have been formally vetted, our aim was also to produce enough non-trivial examples for the automation described in Section 10.

We analyze the Web delegation protocol OAuth Version 2.0 in the Universal Composability (UC) Security framework. We have chosen to only Authorization Code mode of OAuth 2.0. Analysis of the Implicit Grant mode will be covered in an extension.

9.1 Outline

The following is an outline of this section:

Section 9.3 gives a formal definition of the Secure Channel Ideal functionalities and abstraction for SSL based secure channels, which will be used extensively in this section, as well as describe conventions that will be used in defining ideal functionalities.

Section 9.4 defines the OAuth 2.0 Authorization Code Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$. It assumes that the Client (also called Consumer) and Service Provider have globally known names, whereas the User (or user agent) only has local identity based on a userid, password account with these global entities.

Section 9.5 and Fig 14 give a real-world realization of the Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$ using SSL. This implementation is general purpose and does not restrict the User code to be a user agent (i.e. an http browser). This section rigorously proves that this is a universally composable (UC) secure implementation. Later, in Fig 15 we give a refinement of this implementation where the User code is restricted to being a User Agent capable of handling `https` redirections. Further still, in Figs 16 and 17 we give refinements where some of the initial flows are based on hyper-links being sent by authenticated or unauthenticated emails respectively by the Client (Consumer) to the User.

9.2 Security Analysis Synopsis

Salient findings related to security of OAuth 2.0 Authorization Code mode include:

1. The protocol in Fig 15, which we prove to be a secure realization of our Ideal Functionality for OAuth, is more or less the same as defined in the IETF internet draft OAuth Version 2.0 Authorization Code mode [HL10], except for a few fine points noted below.

2. Web-Servers which serve arbitrary Users on the Internet must have public keys for authenticating themselves. Thus, both the Service Provider and the Consumer (Client) must have public keys. Further, for simplicity, assuming that the public key of the Service Provider is attested by a global Certificate Authority (CA), the Public Key of the Client should at least be pre-registered with this globally certified Service Provider (if Client itself does not already have a globally certified public key). If the Client's public key is not globally certified, then its public key must be distributed securely to applications running on User's machine and capable of verifying signatures issued with this public key (in essence, implementing SSL).

3. Given that the Service Provider is required to have a globally certified public key, one can assume that it can handle SSL requests, in particular https requests. Since the Client also has a public key with the public key securely delivered to the User, the User Machine's application must be able to run a protocol with this Client which essentially implements SSL.

4. It is advisable that the login form presented by the Service Provider to the User contain both the Globally certified name of the Service Provider (e.g. Charles Schwab) **and** the Globally known name of the Client (e.g. turboTax) as to whom temporary access is being granted. While this is not that important in cases where the hyper-link to the Service Provider's login page comes from the Client via an SSL connection, but it can be important where the hyper-link to the Service Provider's login page comes via un-authenticated email (see fig 17).

5. In the flow where the Client (Consumer) presents the Authorization Code to the Server to get back an Access Token, the Service Provider should establish that this Client is the same as the one to whose redirect URI the Authorization Code was sent. The correct flows are as follows (or any equivalent flows). During Authorization Grant, when the Server is presented with a Client's redirect URI, a public key of the Client, and a certificate on this public key, the Server must check that this redirect URI belongs to the same entity, by either checking for this information in the certificate or by checking that this fact has been pre-registered. The session id for this session is saved by the Server along with this public key of the Client. Later, when the Client comes back to present the Authorization Code to get the AccessToken, the Server must check that this request is coming from a Client with the same public key. This can be done e.g. by using SSL/TLS two-side certificate checking. This essentially implements the Secure Channel Ideal Functionality (Fig 11).

   We point out that this check is only required if the Client and User have an authentication mechanism (e.g. a userid, password) of their own, which is most likely the case. If the Client never authenticates the User, then because of other security issues, the fine points of the previous paragraph are moot.

6. If the Client authenticates the User, say by a userid, password account (which could be same or different from the account the User has with the Provider), then every fresh

SSL session between the User and the Client must re-do this authentication. Thus, in the protocol we give (Figs 14 and 15), there are two SSL sessions between the User and the Client, and the second can be replaced by just using the first if the first is still active. If however, the first session has expired (and cannot even be refreshed) and a new SSL session must be established, then the authentication of the User by the Client must be re-done in this new SSL session.

9.3 The Secure Channel Ideal Functionality

Cryptographic protocols are inherently multi-party protocols where the parties jointly compute some function. While the issue of liveness of protocols falls in the realm of distributed computing, we are interested here in issues related to privacy (secrecy) of the parties, as well as the related issue of authentication.

These multi-party privacy constraints of a protocol are best **defined** by hypothetically employing an *ideal trusted third party*. In the simplest form of this definitional paradigm, all parties hand their respective inputs to the ideal trusted third party, which then computes the function on these inputs as desired by the protocol being defined, and then hands portions of the output to the individual parties, the portions restricted to what is desired by the protocol.

Note that in the above we assumed that the parties have a secure and persistent tunnel to the ideal third party, but this being just a definition this assumption is not a concern. In addition, if the protocol requires that a party in this multi-party protocol be a particular globally known entity, then we *further assume* that one of these secure tunnels is known to the ideal third party to be connected to that globally known entity. In the real world this can for example be implemented using a public key infrastructure (PKI). For parties that are not globally known, the ideal third party just assumes a persistent connection to some party (e.g. this will guarantee that it gives output to the same party that supplied some input using that tunnel). These issues will be brought up again in the examples below.

In a more general setting, this definitional paradigm also allows the ideal third party to leak certain information to an **Adversary** (without loss of generality, we always assume that there is *only* one monolithic adversary). This maybe a necessary part of the definition as otherwise there may not be any implementation possible in the real world (i.e. a distributed implementation without the ideal third party). In a further generalization, the ideal third party may also take directives and/or inputs from the Adversary.

Finally, to allow a **compositional** definition, we assume that all parties (except the ideal third party, but including the Adversary) are driven by an all encompassing entity called the **Environment**. In other words, the inputs of the parties (possibly related) are actually provided by the Environment so as to cover all possible scenarios. This notion of Environment is *not necessary* to understand ideal functionalities, but it will become important when we discuss compositional security.

Coming to our first example, we discuss the ideal functionality "Secure Channel" described in Fig 11. It is the ideal functionality representing authenticated and encrypted

Ideal Functionality $\mathcal{F}_{\text{SC}}$

$\mathcal{F}_{\text{SC}}$ proceeds as follows, running with parties $P_1, ..., P_n$ and an adversary $\mathcal{S}$.

1. Upon receiving a value (`Establish-session`, sid, P_j, initiator) from some party, P_i, send (sid, `Establish-Session`, P_i, P_j) to the adversary, and wait to receive a value (`Establish-session,` sid, P_i, responder) from P_j. Once this value is received, set a boolean variable `active`. Say that P_i and P_j are the partners of this session.

2. Upon receiving a value (`Send`, sid, m) from a partner P_e, $e \in \{i, j\}$, and if `active` is set, send (`Received,`sid, m) to the other partner and ($sid, P_i, |m|$) to the adversary.

3. Upon receiving a value (`Expire-session`, sid) from either partner, un-set the variable `active`.

Figure 11: The Secure Channels functionality, $\mathcal{F}_{\text{SC}}$

Ideal Functionality $\mathcal{F}_{\text{SSL}}$

The functionality $\mathcal{F}_{\text{SSL}}$ is between a party P_j (the Server) and an unnamed party designated as the client. This models the asymmetric situation when the server has a public idenity but a client who connects to it has no global identity.

1. Upon receiving a value (`Establish-session`, sid, P_j, `initiator`) from some party, say P_i, the functionality first sends the message (sid, `Establish-Session, initiator`, P_i, P_j) to the adversary. If the adversary responds positively the functionality records P_i as the client in this session and sends the message (sid, `Establish−request`) to P_j. When it receives the value (`Establish-session,` sid, `responder`) from P_j in response it sends (sid, `Establish-Session, respondor`, P_j) to the adversary, and it sets the session as `active` between client P_i and server P_j.

2. Upon receiving a value (`Send,`sid, m) from a party P_e, it checks to see if the session sid is `active` and that P_e is either P_i or P_j. It then sends (sid, P_e`,|m|`) to the adversary and when it receives a positive response it sends (`Received,`sid, m) to the other partner in the session.

3. Upon receiving a value (`Expire-session`, sid) from either the client or server in the session it sets state to `inactive` and public delayed informs the other party.

Figure 12: The SSL functionality, $\mathcal{F}_{\text{SSL}}$

channels between two globally known parties, e.g. two parties with public keys in a PKI. Since, the same two parties may be involved in multiple different secure channel sessions

(possibly concurrently), we assume that the two parties have decided on a session id (sid) before hand; in fact this sid may be decided by the Environment as it drives all the parties anyway. We emphasize that this definition of secure channel requires that the parties have globally known names. For instance when party P_i initiates a session with an Establish-session input to the ideal functionality, it is naming a party P_j with which it wants to have the secure channel session. Similarly, the same named party P_j must respond with the Establish-session input while naming P_i. Once, both parties have supplied their Establish-Session inputs, while naming the correct counter-parties, the ideal functionality sets its local state to *active*. From then on, either party can send a message to the other party using the ideal functionality, which only leaks the length of each message to the Adversary (it is possible to have an ideal functionality which does not even reveal the length of the message to the adversary, but implementing such a functionality in the real world may be very inefficient).

Moving on to the next example, the SSL ideal functionality in Fig 12, one notices in contrast to the secure-channel functionality above that only one of the peers is supposed to have a globally known name, and the session can be initiated by any party P, and whose name is never referred to in any messages (initialization or otherwise). The adversary is leaked information about the name being used by this client (e.g. a temporary IP address), which is unavoidable in any real-world implementation.

9.3.1 Conventions for Defining Ideal Functionalities

Delayed Messages When an ideal functionality outputs a value to a party, this delivery of value can be either **public delayed** or **private delayed**. Essentially, since the network in the real-world is assumed to be under adversarial control, one can only guarantee delivery of messages in the real-world if the adversary complies. Thus, any value to be delivered to a party is termed "delayed", which is a short form saying that the functionality requests the Adversary to deliver the value, and if the adversary responds positively, then the value is actually delivered. When the functionality terms the delivery public delayed, it means that the adversary gets to see the actual value, whereas if it is termed private delayed, then the adversary only gets to see the length of the message in bits.

Corruption Each ideal functionality has a function called **Corrupt** which the Adversary can call to declare a globally known entity (i.e. one with a public key in PKI) corrupted. The functionality then records in its local state that the party is corrupted. The functionality may choose to have a different behavior based on whether certain parties are declared corrupted. Normally, this would mean, for example, instantly revealing some internal state of the ideal functionality to the Adversary, as that is what happens in the real-world. Ideally speaking, one would like to have the case that even if some parties in a protocol are corrupted, it does not affect the privacy of other parties, and most ideal functionality would be defined in that fashion. However, one cannot exclude some side effects from happening when some party in the protocol is corrupted.

As an interesting example related to OAuth, suppose that legitimate Clients (or Con-

sumers) are registered with the Service Provider. In fact, under PKI attestation of the URIs of the Clients, the Service Provider just needs to know the global identity of these Clients. If a User agent contacts (under SSL) a Service Provider with a URI of a Client (attested under PKI to be associated with a globally known name P_c), and further if the User Agent manages to provide correct userid_A, password_A to the Service Provider under the same SSL session, then the Provider can assuredly provide the client P_c information related to the account userid_A. Now consider the situation where the password_A was easy to guess, and hence the User Agent was being driven by an illegitimate user (Adversary). If the Client was not authenticating the User on its own, then the Adversary would end up getting information related to userid_A (via the Client). But, if the Client was also authenticating the User under possibly a different userid_B, password_B, and if this password was tough to guess for the Adversary, then the Client will never open a session with this illegitimate user. However, a different Client P_c' which is a pre-registered Client may be malicious and may join forces with the Adversary. In this case, as soon as the illegitimate user presents userid_A, password_A, P_c' to the Service Provider, the information related to userid_A is essentially given to the Adversary by the Provider. The ideal functionality would do exactly that, but it needs to be informed that P_c' has been corrupted (i.e. it has joined forces with the Adversary).

Session Id The session identifier or *sid* is assumed to be unique (but non-secret) for each instantiation of the real-world protocol or the ideal functionality. Usually, in the Universal composability paradigm [Can01], the *sid* is assumed to be determined by a wrapper protocol (hence the Environment) between all legitimate parties, and this pre-determined *sid* is passed along with other inputs by the Environment to all legitimate parties of the protocol. However, this wrapper protocol could lead to an inefficient complete protocol, as it may take additional flows. Thus, in our definition of OAuth's ideal functionality, we have minimized any such wrapper protocol for determination of *sid*, so that the resulting implementation is in line with practical implementations of OAuth. One implication of this is that the ideal functionality has many more flows than one would expect. For example, the Service Provider is not pre-determined by the *sid*, and is actually passed as an input by the Client. Similarly, the Client is not pre-determined by the *sid*, and the Client's global identity is passed to the Provider (with possible Adversarial manipulation of this identity) in the ideal functionality.

9.4 The OAuth Ideal Functionality

Before we embark on giving a definition of the OAuth Ideal Functionality, we paraphrase the following abstract from the OAuth V2 Internet draft ([HL10]):

"*The OAuth 2.0 authorization protocol enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.*"

We will focus mainly on the first kind of delegation which requires an approval interaction. In the ideal functionality definition, we will refer to the three parties named above as **User** (resource owner), **Provider** (HTTP Service), and **Consumer** (third-party). Moreover in this section, we will assume that the Provider and the Consumer have globally known names, e.g. public keys with certified domain names in PKI. Only the User will *not* be assumed to have a global name. The case where the Consumer has a public key which is not certified by a global CA is dealt with in a later section.

In particular, the most interesting and wide-spread case arises where the User has a $\langle \text{userid}, \text{password} \rangle$ account with a Provider, which needs to be delegated to the Consumer. The basic idea of the definition can be summarized as follows: A User starts of using a software provided by a Consumer P_c, say as a web site with PKI certified public key. To model this, the User calls the ideal functionality with an input message $(sid, \texttt{initiate}, P_u, P_c)$. Here sid is a non-secret but unique session identifier which is chosen by the User or Environment (wrapping the User). Note that P_u may just be a temporary identifier, e.g. a temporary IP address. The Consumer, at some point decides that it needs access to data of the User held at a Provider P_s, again P_s being a web site with a PKI certified public key (see Fig 13 for a formal description and Fig 18 for a pictorial depiction). The ideal functionality then forwards the name P_s to the User, who in response provides a userid and password to the ideal functionality (corresponding to its account at P_s). The ideal functionality then provides the userid and the name P_c to P_s. At this point, P_s may abort, if for example it does not wish to provide data to P_c, or if it deems userid to be invalid. Otherwise, it responds to the ideal functionality with the pre-registered pw' corresponding to the userid provided Note that, all parties are really being driven by wrapper software around these parties, which we treat as a single monolithic Environment. The main reason for treating the Environment as monolithic is that one *cannot* assume that the individual wrapper software are *not* talking to each other via some other protocols and in the process leaking information. Thus, the universal composability (UC) paradigm lets one analyze the security of the protocol even if the wrapper software are buggy, or downright maliciously collusive.

The ideal functionality next checks the two passwords for equality (the one supplied by the User and the one returned by the Provider). If they are not equal, the session is aborted, and the Adversary is also leaked this information (as it is highly inefficient to not leak this information in the real-world). If they are equal then the functionality sets a local status variable, say authentication status to valid, and this status is also revealed to the Adversary.

Next, the ideal Functionality issues a randomly generated AccessToken to the Provider P_s and Consumer P_c, but only at the Adversary's directive, i.e. the Adversary can cause denial of service. This common and secret AccessToken may be used by the two parties to communicate any information related to the account corresponding to userid. Note again that the wrapper software of the Provider is given the userid and the AccessToken, and if the wrapper software is buggy, there are no security guarantees. The ideal functionality only guarantees that a userid has been provided to the Provider, and further that the password the Provider gave for this userid is same as the password provided by some arbitrary party P_u, and that the Provider shares the AccessToken with a globally known entity P_c, the

Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$

Participants: An arbitrary User P_u, Service Provider P_s, Consumer P_c, and Adversary $\mathcal{S}$ interact with functionality $\mathcal{F}_{\text{OAUTH}^*}$ (or $\mathcal{F}$).

Status Variables: Deflection Status (default: normal), Usurpation Status (default: normal), Authentication Status (default: empty).

Initiate: On receiving an input message $(sid, \texttt{initiate}, P_u, P_c)$ from P_u, output (public delayed) the message $(sid, \texttt{params}-\texttt{req})$ to P_c. On receiving a reply $(sid, \texttt{params}, P_s)$ from P_c, output (public delayed) the message $(sid, \texttt{params}, P_s)$ to P_u.

On receiving a response $(sid, \texttt{credentials}, userid, pw)$ from P_u, output $(sid, \texttt{initiate}-\texttt{req}, userid, P_c)$ to adversary $\mathcal{S}$. On receiving a response $(sid, \texttt{initiate}-\texttt{req}, userid', P_c')$ from adversary S, output $(sid, \texttt{initiate}-\texttt{req}, userid', P_c')$ to P_s. Further, if $userid'$ is not the same as $userid$ or P_c' is not the same as P_c, set the *deflection status* as `deflected`.

Next, on receiving a response $(sid, \texttt{password}, pw')$ from P_s, record pw' locally. If the *deflection status* is not set as `deflected`, then check if $pw = pw'$, and if the two passwords are not the same then set the *authentication status* as `aborted` and output $(sid, \texttt{abort}-\texttt{oauth})$ to the Adversary. Otherwise (if the passwords are same), set *authentication status* as `valid` and output $(sid, \texttt{initiate}-\texttt{oauth})$ to the Adversary. If the deflection status is set as `deflected`, skip the above steps and wait for a TestPwd call from $\mathcal{S}$.

Instead of responding with a pw', P_s may instead respond with $(sid, \texttt{bad}-\texttt{params})$, in which case the functionality sets the *authentication status* as `aborted` and outputs $(sid, \texttt{bad}-\texttt{params})$ to $\mathcal{S}$.

The following call may be made by the Adversary $\mathcal{S}$ (and only $\mathcal{S}$).

TestPwd: On receiving a message $(sid, \texttt{testpwd}, pw'')$ from S: If pw' is not recorded locally, then ignore this call. Else, if $pw'' = pw'$ then set usurpation status as `compromised`, otherwise set usurpation status as `interrupted`. Report the usurpation status to $\mathcal{S}$. In either case reset *authentication status* to empty. Note that deflection status is immaterial here, and $\mathcal{S}$ can make this call even if deflection status is normal.

The following two calls may also be made by the Adversary $\mathcal{S}$ (and only $\mathcal{S}$) in any order.

Issue Access Token to Consumer: On receiving a message $(sid, \texttt{IssueKey2Consumer}, k)$ from S, if the authentication status is set as `valid` then output a randomly generated *AccessToken* to P_c. Else, if the usurpation status is set as `compromised`, output k (provided as a parameter by the Adversary) to P_c. In all other cases, obtain fresh $r \leftarrow \$$, and output r to P_c .

Issue Access Token to Service Provider: On receiving a message $(sid, \texttt{IssueKey2SP}, k)$ from S: if the authentication status is set as `valid` then output the same *AccessToken* as above to P_s. Else, if the usurpation status is set as `compromised`, *and* P_c' is same as P_c or P_c' is corrupted, output k (provided as a parameter by the Adversary) to P_s. In all other cases, obtain fresh $r \leftarrow \$$, and output r to P_s .

Send: On receiving an input $(sid, \texttt{Send}, m)$ from $\hat{P}$ which is either P_u or P_c, check if the authentication status is set as `valid`. If so, send $(sid, \hat{P}, |m|)$ to Adversary $\mathcal{S}$, and after receiving a positive deliver response from $\mathcal{S}$, output $(sid, \texttt{Received}, m)$ to the counter-party of $\hat{P}$ (i.e. P_c or P_u resp.). If instead, the usurpation status is set as `compromised` and if `Send` originates from P_c (i.e. $\hat{P}$ is P_c), and $P_c' == P_c$, then send $(sid, \hat{P}, m)$ to adversary $\mathcal{S}$ (i.e. the Adversary gets the message m). In all other cases, send $(sid, \hat{P}, |m|)$ to $\mathcal{S}$ and take no further action.

Figure 13: A functionality for delegation with explicit key exchange

identity P_c being supplied by the same party P_u.

The ideal functionality also provides a capability for the User and the Consumer to send messages to each other secretly. The authentication of messages originating at P_c do not need any particular mention since P_c has a global public key, but what is noteworthy is that

the delivery of these messages is *only* to the User who provided the correct password to the userid submitted to the Provider.Note that if the Adversary does not deflect the protocol, and change userid to some other userid', then the account corresponds to userid as provided by P_u. If however, the Adversary (say posing as another P'_u) hijacks (or deflects) the flow of the protocol by injecting a different userid', then OAuth will continue in normal fashion only if the Adversary provides the correct password for userid', and then it will get results corresponding to this account.

The OAuth Ideal Functionality is defined in detail in Figure 13. In this detailed definition, one notices that the Adversary makes the "Issue Key" calls. As already mentioned, this makes sense as the network is assumed to be insecure, and hence the Adversary is assumed to control the network. Thus all network flow is directed by the Adversary, including the issuing of keys. Note that the Adversary does not control what keys are delivered (unless it has compromised the session), but only control when and if the keys are delivered.

There are certain other intricacies related to **password-based authentication**, which can be ignored in a first reading. Since the password in password-based accounts is usually human memorizable, it can not be assumed to have full 128 bit entropy (when talking about 128-bit security for the protocol). Thus, the ideal functionality we define must not guarantee 128-bit security, since any implementation of such an ideal functionality would require random 128 bit passwords.

However one can define an ideal functionality which guarantees that an Adversary can only by-pass the 128-bit security by performing online guessing attacks. Thus, other than these guessing attacks (which are un-avoidable), 128-bit security is guaranteed by the ideal functionality. So, how does one define an ideal functionality where such guessing attacks are feasible? The functionality allows the Adversary to call it with a guess of the password, and if the guess is correct, the Adversary is allowed to set shared keys of its choosing (basically giving it access to the secure channel being setup), and if the guess is incorrect then the session is considered interrupted, and the legitimate parties (essentially) abort.

9.5 Implementation of Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$

The implementation of the Ideal Functionality $\mathcal{F}_{\text{OAUTH}^*}$ assuming ideal functionalities for SSL and Secure Channels is given in Figure 14. This implementation does not assume that the User program is just a User Agent (i.e. a browser implementing http). However, care has been taken to make this implementation easily refined into one where the User code can just be replaced by a browser that supports redirection. Indeed, such a implementation is given in Fig 15. Further, another implementation where a User may instead be sent a hyper-link of the Provider in an authenticated email (instead of a response to an https request) from the Consumer is given in Fig 16. This implementation is a further refinement of the one in Fig 15. Next, an implementation is given in Fig 17, where the hyper-link is sent via an unauthenticated channel, e.g. posted on a bulletin board. this implementation is a further refinement of the previous ones. Special attention should be paid to the notes at the bottom of the figures.

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using $\mathcal{F}_{\text{SSL}}$ and $\mathcal{F}_{\text{SC}}$

Participants: The User Agent (P_u), Service Provider (P_s), Consumer (P_c), and Adversary $\mathcal{A}$.

Initiate: On receiving $(sid, \texttt{initiate}, P_u, P_c)$ from $\mathcal{Z}$, P_u initiates an SSL session with session id $sid^{\text{UC1}}_{\text{SSL}}$ with P_c and sends the message $(sid, \texttt{params-req}, P_u, P_c)$. On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over SSL session $sid^{\text{UC1}}_{\text{SSL}}$, P_c outputs $(sid, \texttt{params-req}, P_u, P_c)$. On receiving $(sid, \texttt{params}, P_s)$ from $\mathcal{Z}$, P_c sends the message $(sid, \texttt{params}, P_s)$ to P_u using SSL session $sid^{\text{UC1}}_{\text{SSL}}$.

On receiving $(sid, \texttt{params}, P_s)$ over $sid^{\text{UC1}}_{\text{SSL}}$, P_u outputs $(sid, \texttt{params}, P_s)$ to $\mathcal{Z}$. On receiving $(sid, \texttt{credentials}, userid, pw)$ from $\mathcal{Z}$, P_u initiates an SSL session with P_s with session id $sid^{\text{UP}}_{\text{SSL}}$ and sends $(sid, \texttt{initiate-req}, userid, pw, P_c)$. On receiving $(sid, \texttt{initiate-req}, userid, pw, P_c)$ over the SSL session $sid^{\text{UP}}_{\text{SSL}}$, P_s queries the environment with $(sid, \texttt{initiate-req}, userid, P_c)$. The environment responds with $(sid, \texttt{password}, pw')$. P_s tests whether $pw = pw'$. If the check succeeds, P_s generates a random $AuthCode$, records $(sid, P_s, P_c, Authcode)$, and sends the message $(sid, \texttt{authgrant}, AuthCode)$ over $sid^{\text{UP}}_{\text{SSL}}$.

If the environment responds to P_s with $\texttt{bad-params}$, P_s just aborts.

On receiving $(sid, \texttt{authgrant}, Authcode)$ over $sid^{\text{UP}}_{\text{SSL}}$, P_u sends $(sid, \texttt{authgrant}, Authcode)$ to P_c over a *new* SSL session $sid^{\text{UC2}}_{\text{SSL}}$.

On receiving $(sid, \texttt{authgrant}, Authcode)$ over $sid^{\text{UC2}}_{\text{SSL}}$, P_c initiates a secure channel $\mathcal{F}_{\text{SC}}$ with P_s with sid $sid^{\text{CS}}_{\text{SC}}$ and sends P_s the message $(sid, \texttt{authgrant}, P_c, Authcode)$. On receiving $(sid, \texttt{authgrant}, P_c, Authcode)$ over $sid^{\text{CS}}_{\text{SC}}$, P_s checks it against the recorded information, and makes sure that the Secure channel is with the same global entity P_c as recorded earlier. If the check succeeds then it generates $AccessToken$ and sends $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\text{CS}}_{\text{SC}}$. It also outputs $(sid, \texttt{keyCS}, AccessToken)$ to the environment.

On receiving $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\text{CS}}_{\text{SC}}$, P_c outputs $(sid, \texttt{keyCS}, AccessToken)$ to the environment.

Send: On receiving $(sid, \texttt{send}, m)$ from the environment, P_u sends the message $(sid, \texttt{send}, m)$ to the peer over SSL session $sid^{\text{UC2}}_{\text{SSL}}$. On the other hand, on receiving $(sid, \texttt{send}, m)$ from the environment, P_c checks that it has obtained an AccessToken from P_s, and only then it sends the message $(sid, \texttt{send}, m)$ to the peer over SSL session $sid^{\text{UC2}}_{\text{SSL}}$.

Notes:

1. P_u can use the older SSL session $sid^{\text{UC1}}_{\text{SSL}}$ instead of a fresh $sid^{\text{UC2}}_{\text{SSL}}$, if the former is still active.
2. If the Consumer also requires a password-based authentication of the User (possibly with a different userid, password than the one which the User has with the Provider), then if a fresh $sid^{\text{UC2}}_{\text{SSL}}$ is initiated, then this password-based authentication by the Consumer must take place afresh as well.

Figure 14: OAuth v2 Authorization Grant Flow

Theorem 4 *The implementation realizes the functionality $\mathcal{F}_{\text{OAUTH}^*}$.*

Proof: We will show that for every probabilistic polynomial time (PPT) adversary $\mathcal{A}$, there exists a PPT adversary $\mathcal{A}$', such that the ensembles, corresponding to the view of the environment in the experiment where it interacts with the adversary $\mathcal{A}$' involved with the ideal functionality, and the experiment where it interacts with the adversary $\mathcal{A}$ involved with the real implementation, are indistinguishable. The adversary $\mathcal{A}$' in the ideal world is obtained by composing a simulator $\mathcal{S}$ with $\mathcal{A}$ itself, where $\mathcal{S}$ using access to ideal functionality (as ideal world adversary) simulates the real-world to $\mathcal{A}$. Further, $\mathcal{S}$ will ensure that the interaction of $\mathcal{F}_{\text{OAUTH}^*}$ with environment is also indistinguishable with interaction of real parties with the environment. Note that in the ideal world experiment, the parties P_u, P_c, P_s

run as dummies and just pass back and forth the messages between $\mathcal{Z}$ and $\mathcal{F}_{\text{OAUTH}^*}$, whereas in the real world, the parties P_u, P_c, P_s are running the protocol π. Thus to simulate the real world to $\mathcal{A}$, $\mathcal{S}$ will need to simulate the real parties P_u, P_c, P_j to A. We will show that S is able to do so (using access to ideal functionality$\mathcal{F}_{\text{OAUTH}^*}$), and hence as far as the environment Z is concerned the ideal world and the real world are indistinguishable.

We will show later how $\mathcal{S}$ simulates SSL and SC, but for now we deal with how $\mathcal{S}$ simulates the real world parties. To begin with, in both worlds, the environment $\mathcal{Z}$ sends the input $(sid, \texttt{initiate}, P_u, P_c)$ to P_u. In the real-world, this prompts P_u to Establish an SSL session by calling the ideal functionality $\mathcal{F}_{\text{SSL}}$, which in turn calls the Adversary $\mathcal{A}$ for a response. This call to $\mathcal{A}$ needs to be simulated by $\mathcal{S}$, but since in the ideal world $\mathcal{S}$ gets called by $\mathcal{F}_{\text{OAUTH}^*}$ as well, $\mathcal{S}$ can at that point simulate a call to $\mathcal{A}$. If $\mathcal{A}$ responds positively in the real-world, $\mathcal{S}$ in the ideal world calls $\mathcal{F}_{\text{OAUTH}^*}$ positively (note $\mathcal{S}$ is using $\mathcal{A}$ as a blackbox). Note that this leads to an output via P_c to the environment of value $(sid, \texttt{params-req}, P_u, P_c)$ in both worlds.

Next, in both worlds, the environment $\mathcal{Z}$ sends input $(sid, \texttt{params}, P_s)$ to P_c. In the real world, this prompts P_c to **send** a message via the same SSL session to P_u, which is promptly output back to the environment. In the ideal world, the simulator $\mathcal{S}$ gets notified by the ideal functionality $\mathcal{F}_{\text{OAUTH}^*}$ of this input from P_c (as it is being sent to P_u public delayed), and hence $\mathcal{S}$ can simulate a call to $\mathcal{A}$ regarding the size of the message in bits (which is what $\mathcal{F}_{\text{SSL}}$ does in the real world). Further note that the message is also output to the environment in the ideal world.

Similarly, as long as none of the SSL session establishments are replaced by the Adversary by its own establishment using corrupted parties, the simulation can be done by $\mathcal{S}$.

Now consider the cases where one or more of the SSL sessions $sid^{\text{UC1}}_{\text{SSL}}$, $sid^{\text{UC2}}_{\text{SSL}}$, $sid^{\text{UP}}_{\text{SSL}}$ may be initiated by a corrupt principal (P'_u). We don't have to consider the secure channel $sid^{\text{CS}}_{\text{SC}}$ because a secure channel authenticates both the peers, and we do not consider corruption of the client and/or the provider here.

Case $sid^{\text{UC1}}_{\text{SSL}}$ is initiated by a corrupt principal: First we see how an Adversary $\mathcal{A}$ in the real-world accomplishes this. Assume that the Adversary has managed to corrupt a user (lets call it P'_u; infact the user P'_u in this case may just be the Adversary). From now on we will identify P'_u with $\mathcal{A}$, which goes well with our convention that there is only one monolithic Adversary. Next, when the legitimate user P_u makes an Establish Session call to $\mathcal{F}_{\text{SSL}}$, the Adversary $\mathcal{A}$ does not respond with a positive response, and hence in a sense that SSL connection never takes place. Instead, the Adversary (via P'_u) initiates a new SSL establish session call to another instance of $\mathcal{F}_{\text{SSL}}$ naming the same consumer P_c as server.

since the Adversary is making this Establish session call to $\mathcal{F}_{\text{SSL}}$, the Simulator which is treating $\mathcal{A}$ as a black box can see this call being issued. Similarly, when $\mathcal{A}$ (via P'_u) sends a message $(sid, \texttt{par-req})$ by calling the Send function of $\mathcal{F}_{\text{SSL}}$, again $\mathcal{S}$ gets to see this call in the plain, and checks for integrity of the message, and if so, just responds positively to the ideal functionality $\mathcal{F}_{\text{OAUTH}^*}$'s public delayed output to P_c. Thus, in

both worlds P_c outputs the same value `par-req` to $\mathcal{Z}$. Note that in the real-world P_c has no idea whether P'_u is legitimate or not, and hence as long as the message was of proper syntax, it will output it to $\mathcal{Z}$.

Case $sid^{\texttt{UP}}_{\text{SSL}}$ is initiated by a corrupt principal: Again, as in the previous case, the Adversary may not respond positively to a legitimate SSL establish-session request, and instead start its own SSL session with the Provider P_s. However, since it is $\mathcal{A}$ that sends a message $(sid, \texttt{initiate-req}, userid'', pw'', P''_c)$ using this SSL session, $\mathcal{S}$ is able to see the message. If the message is not syntactically correct then $\mathcal{S}$ just stops. $\mathcal{S}$ then sends the message $(sid, \texttt{initiate-req}, userid'', P''_c)$ to $\mathcal{F}_{\text{OAUTH}^*}$ which the latter is expecting. If $userid$ or P_c have been altered from what $\mathcal{F}$ sent to $\mathcal{S}$, then F sets its deflection status to deflected. Note that this can only happen if $\mathcal{A}$ started its own SSL session, although $\mathcal{A}$ may choose to keep $userid$ and P_c same (which although not revealed to $\mathcal{A}$ in the real-world protocol, may still be easily guess-able).

Next, the environment receives the output $(sid, \texttt{initiate-req}, userid'', P''_c)$ in both the worlds. It responds with $(sid, \texttt{password}, pw')$ if the environment (i.e. P_s's wrapper software) determines that $userid''$ and P''_c are valid parameters (or it may respond with `bad-params`; this case is easy to handle and we skip this case). Note that if P''_c is different from P_c, but still a pre-registered Consumer, then the environment is likely to pass it as a valid parameter. Further, since the SSL session is being initiated by the Adversary, the initial user P_u is out of the picture. There are two sub-cases depending on the deflection status of $\mathcal{F}$.

normal: In this case, on the input pw' from P_s, the functionality F sets its authentication status based on whether pw' is same as pw (supplied by P_u). However, this is not what happens in the real-world, where pw' is compared with pw'', the latter being supplied by $\mathcal{A}$. Thus $\mathcal{S}$ must call `TestPwd` function of $\mathcal{F}$ with pw'', which leads to F ignoring (or clearing) the authentication status, and instead setting the usurpation status based on whether pw' is same as pw''. (Note that $\mathcal{S}$ had obtained pw'' when $\mathcal{A}$ issued a send message call to $\mathcal{F}_{\text{SSL}}$ – see previous paragraph.) Now, both the real-world and the ideal-world are in sync.

`deflected`**:** In this case, on the input pw' from P_s, the functionality F does not set its authentication status anyway, so $\mathcal{S}$ just needs to call F's `TestPwd` function with pw''.

Next in the real world, P_s sends the message $(sid, \texttt{authgrant}, AuthCode)$ over $sid^{\texttt{UP}}_{\text{SSL}}$ only if passwords matched, in which case in the ideal world $\mathcal{S}$ (since it s informed the usurpation status) just generates random Authcode of its own and delivers it to $\mathcal{A}$.

Note that, in the real-world the Adversary must initiate its own SSL session UC2 with the Consumer, otherwise the Authcode it obtained from the Provider cannot be delivered back to the Provider via the consumer. If $\mathcal{A}$ indeed starts its own SSL session with P_c, then since it will call the SSL session with a Send Authcode, the Simulator can check if this Authcode is same as the one it generated for it.

Case $sid^{\texttt{CS}}_{\text{SC}}$ is initiated by a corrupt party P'_c. In the real-world P_s checks that this P'_c is same as the parameter sent to it in $sid^{\texttt{UP}}_{\text{SSL}}$, or more precisely its replacement initiated by Adversary $\mathcal{A}$. Thus, P_s only continues with the protocol in the real-world if $sid^{\texttt{UP}}_{\text{SSL}}$ was replaced by Adversary, which had passed a P'_c then and which was validated by the Environment (i.e. the environment did not respond with `bad−params`. In such a scenario, recall that the ideal-world has set its deflection status as deflected (note corrupt P'_c is not same as P_c). This means that the authentication status is empty. Now, if in the real-world P'_c also happens to report the correct Authcode in $sid^{\texttt{CS}}_{\text{SC}}$, then P_s and P'_c (i.e. $\mathcal{A}$) will end up sharing a common random AccessToken. To emulate this situation, $\mathcal{S}$ in the ideal world just calls Issue AccesToken to Provider with a randomly chosen k. This would lead to setting the AccessToken of the Provider to k if usurpation status was `compromised` (i.e. $pw'' = pw'$) and P'_c is corrupted (which it is). The simulator $\mathcal{S}$ also sends a message to P'_c (i.e. $\mathcal{A}$) $(\texttt{accesstoken}, k)$ under the appropriated $sid^{\texttt{CS}}_{\text{SC}}$, thus completely emulating the real-world.

Case $sid^{\text{UC2}}_{\text{SSL}}$ is initiated by a corrupt principal: If in the real-world the first two SSL sessions were legitimate, and only this one is initiated by $\mathcal{A}$ then $\mathcal{A}$ can provide the correct Authcode to Consumer only with negligible probability, as Authcode is generated randomly. Thus, in this case in the real-world, the Provider and hence the Consumer will not output a common and random AccessToken. Since, the Simulator $\mathcal{S}$ knows that the first two SSL sessions were legitimate, and this one is not, it just does not make the calls to $\mathcal{F}_{\text{OAUTH}^*}$'s Issue Access Token functions.

If on the other hand, the SSL session between User and Provider was taken over by the Adversary, then, we know from the previous case that the Adversary knows the usurpation status, and in case of compromised status, has provided a random Authcode of its own. If $\mathcal{A}$ in this corrupted SSL session sends the same Authcode to P_c, the $\mathcal{S}$ gets to see that, and hence it means in the real world since P_c and P_s behave honestly, P_s will end up issuing valid (and randomly generated) AccessToken to P_c. So, $\mathcal{S}$ just makes the Issue AccessToken calls for both Consumer and Provider with the same randomly chosen value k. Thus, the view of the Environment in both the worlds will be same.

As for the $(sid, \texttt{Send}, m)$ values given by the Environment to P_c in the real world, note that P_c only sends it to the peer if it had obtained an AccessToken, which is only possible if if the password pw'' the Adversary had provided matched the password P_s had produced, which means the usurpation status is `compromised`. Thus in this situation, if this last SSL session was started by the Adversary, in the real world the Adversary would get the message m, and the same would hold in the ideal world.

□

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using HTTPS Redirection

Participants: An arbitrary User Agent P_u, a Service Provider P_s, a Consumer P_c, and Adversary $\mathcal{A}$. The Service Provider and Consumer are assumed to have SSL supporting URIs with PKI certificates.

Initiate: On receiving an input $(sid, \texttt{initiate}, P_u, P_c)$ from the environment, P_u initiates an **https** session with session id $sid^{\texttt{UC1}}_{\text{SSL}}$ with the URI of P_c and query parameters $(sid, \texttt{params-req}, P_u, P_c)$.

On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over https $sid^{\texttt{UC1}}_{\text{SSL}}$ which it outputs to the environment, P_c obtains the input $(sid, \texttt{params}, P_s)$ from the environement. Then P_c responds with $(sid, \text{redirect uri:}P_s, \texttt{client-params})$ to P_u using SSL session $sid^{\texttt{UC1}}_{\text{SSL}}$. Since P_s is a globally known entity, the Consumer P_c can obtain a valid https URI of P_s. The query parameters called `client-params` includes the https redirection URI of P_c itself, and other credentials of P_c. Since the response is a redirection https URI, the user agent P_u automatically initiates an SSL session with P_s with session id $sid^{\texttt{UP}}_{\text{SSL}}$ and query parameter $(sid, \texttt{client-params})$.

On receiving this message, P_s responds to P_u over $sid^{\texttt{UP}}_{\text{SSL}}$, with a login-password form, to which the user agent responds by outputting the form along with P_s's identity to the environment (which in this case is possibly just a human User along with a certificate checker). The Environment (or the human) responds with a $userid$ and password pw, which the User agent forwards to P_s using $sid^{\texttt{UP}}_{\text{SSL}}$. Next, P_s queries the environment with the received userid along with `client-params` (since P_c is also a globally known entity, this is just equivalent to outputting the identifier P_c itself), to obtain a password pw' corresponding to this userid (or a `bad-params` response). On pw validation (i.e. $pw = pw'$), P_s generates a fresh random number $Authcode$, and saves $(sid, \texttt{initiate}, P_s, P_c, Authcode)$ in its local memory, and responds over $sid^{\texttt{UP}}_{\text{SSL}}$ with the redirect URI of P_c (obtained from `client-params`) and query parameter $(sid, \texttt{authgrant}, AuthCode)$.

Since the user-agent P_u receives a redirect URI, which is an https URI of P_c, it automatically starts a new SSL session $sid^{\texttt{UC2}}_{\text{SSL}}$ with P_c, over which it sends query paramter $(sid, \texttt{authgrant}, AuthCode)$. On receiving $(sid, \texttt{authgrant}, Authcode)$ over $sid^{\texttt{UC2}}_{\text{SSL}}$, P_c initiates a secure channel $\mathcal{F}_{\text{SC}}$ with P_s with sid $sid^{\texttt{CS}}_{\text{SC}}$ and sends P_s the message $(sid, \texttt{authgrant}, P_c, Authcode)$.

On receiving $(sid, \texttt{authgrant}, P_c, Authcode)$ over $sid^{\texttt{CS}}_{\text{SC}}$, P_s checks it against the recorded information. If the check succeeds then it generates a random $AccessToken$ and sends $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\texttt{CS}}_{\text{SC}}$. It also outputs $(sid, KeyCS, AccessToken)$ to the environment.

On receiving $(sid, \texttt{accesstoken}, AccessToken)$ over $sid^{\texttt{CS}}_{\text{SC}}$, P_c outputs $(sid, KeyCS, AccessToken)$ to the environment.

Send: This is same as in Fig 14.

Notes:
1. The User agent may check if the rediection URI of P_c is same as the URI P_c it used in SSL session $sid^{\texttt{UC1}}_{\text{SSL}}$, and if that session is still alive, it can use that same session instead of a new $sid^{\texttt{UC2}}_{\text{SSL}}$.

Figure 15: OAuth v2 Authorization Grant Flow

9.6 Summary

This section described the modeling of the key web security protocol OAuth2.0. We were able to write down a proof of correctness of an important mode of operation of this protocol. This is timely since the protocols is being considered for standardization and a formal proof will give implementers the right assurance. Our analysis has derived specific security recommendations which should be incorporated as implementation guidelines for secure realization. Besides this, we have demonstrated again that UC can be used outside of the domain of purely cryptographic protocols.

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using Authenticated E-mail and HTTPS Redirection

Participants: An arbitrary User Agent P_u, a Service Provider P_s, a Consumer P_c, and Adversary $\mathcal{A}$. The Service Provider and Consumer are assumed to have SSL supporting URIs with PKI certificates.

Initiate: On receiving an input $(sid, \texttt{initiate}, P_u, P_c)$ from the environment, P_u initiates an **https** session with session id $sid^{\texttt{UC1}}_{\text{SSL}}$ with the URI of P_c and query parameters $(sid, \texttt{params-req}, P_u, P_c)$.

On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over https $sid^{\texttt{UC1}}_{\text{SSL}}$ which it outputs to the environment, P_c obtains the input $(sid, \texttt{params}, P_s)$ from the environement. Then P_c responds with $(sid, \text{uri}{:}P_s, \texttt{client-params})$ to P_u using $\mathcal{F}_{\text{AUTH}}(P_c)$, Since P_s is a globally known entity, the Consumer P_C can obtain a valid https URI of P_s. The query parameters called `client-params` includes the https redirection URI of P_c itself, and other credentials of P_c. The user agent P_u on receiving this authenticated message from P_c initiates an SSL session with P_s (using the supplied uri of P_s) with session id $sid^{\texttt{UP}}_{\text{SSL}}$ and query parameter $(sid, \texttt{client-params})$.Technically, the User must click on the supplied link, but we will assume the worst case that the user always clicks on the link.

On receiving this message, P_s responds to P_u over $sid^{\texttt{UP}}_{\text{SSL}}$, with a login-password form, and the rest of the implementation is same as in Fig 15

Notes:

1. The same notes as in Fig 15 apply here.
2. The ideal functionality $\mathcal{F}_{\text{AUTH}}$ is defined in the appendix. It essentially, delivers a message only if the sender is P_c, thus guaranteeing the receiver that the message received came from P_c. This, for example, can be the case where the User has a trusted email server, and the e-mail received has a certified signature of P_c.
3. Note that session $sid^{\texttt{UC1}}_{\text{SSL}}$ need not be an SSL session, but $sid^{\texttt{UC2}}_{\text{SSL}}$ must be an SSL session.

Figure 16: OAuth v2 Authorization Grant Flow in Email Settings

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using Bulletin Board and HTTPS Redirection

Participants: An arbitrary User Agent P_u, a Service Provider P_s, a Consumer P_c, and Adversary $\mathcal{A}$. The Service Provider and Consumer are assumed to have SSL supporting URIs with PKI certificates.

Initiate: On receiving an input $(sid, \texttt{initiate}, P_u, P_c)$ from the environment, P_u initiates an **https** session with session id $sid^{\texttt{UC1}}_{\text{SSL}}$ with the URI of P_c and query parameters $(sid, \texttt{params-req}, P_u, P_c)$.

On receiving $(sid, \texttt{params-req}, P_u, P_c)$ over https $sid^{\texttt{UC1}}_{\text{SSL}}$ which it outputs to the environment, P_c obtains the input $(sid, \texttt{params}, P_s)$ from the environement. Then P_c responds with $(sid, \text{uri}{:}P_s, \texttt{client-params})$ to P_u using an un-authenticated and unencrypted channel. Since P_s is a globally known entity, the Consumer P_C can obtain a valid https URI of P_s. The query parameters called `client-params` includes the https redirection URI of P_c itself, and other credentials of P_c. The user agent P_u on receiving this un-authenticated message from P_c, initiates an SSL session with P_s (using the supplied uri P_s) with session id $sid^{\texttt{UP}}_{\text{SSL}}$ and query parameter $(sid, \texttt{client-params})$.Technically, the User must click on the supplied link, but we will assume the worst case that the user always clicks on the link.

On receiving this message, P_s first checks that the `client-params` has the URI of some P_c along with its Application name which are globally known (e.g. via PKI). Next, P_s responds to P_u over $sid^{\texttt{UP}}_{\text{SSL}}$, with a login-password form *along with the Application name of P_c displayed on the form*, to which the user agent responds by outputting the form along with P_s's identity to the environment (which in this case is possibly just a human User, along with a PKI certificate checker). We will assume here that the User Agent gets human assist in verifying the the P_c displayed on the form is the same as the P_c initially supplied to the User-agent in the `initiate` call.More rigorously, this mode would require a different ideal functionality where both P_s and P_c are provided in the `params output to` P_u`, instead of` P_u `providing` P_c `in the initiate call.`

The Environment (or the human) responds with a *userid* and password *pw*, which the User agent forwards to P_s using $sid^{\texttt{UP}}_{\text{SSL}}$. Rest of the implementation is same as in Fig 15

Notes: 1. The same notes as in Fig 16 apply here.

Figure 17: OAuth v2 Authorization Grant Flow in Bulletin Board Settings

Ideal Functionality Flow $\mathcal{F}_{\text{OAUTH}^*}$

An example flow of the Ideal World Implementation involving $\mathcal{F}_{\text{OAUTH}^*}$, with Adversary mostly responding positively. The symbol $\mathcal{Z}$ stands for the Environment, $\mathcal{F}$ for the ideal functionality, and $\mathcal{S}$ for the Adversary.

$\mathcal{Z}$	$\xrightarrow{\texttt{initiate},P_u,P_c}$	P_u	$\longrightarrow$	$\mathcal{F}$	$\longrightarrow$	$\mathcal{S}$		
					$\xleftarrow{+ve}$			
					$\longrightarrow$	P_c	$\longrightarrow$	$\mathcal{Z}$
		$\mathcal{S}$	$\xleftarrow{+ve}$	$\mathcal{F}$	$\longleftarrow$	P_c	$\xleftarrow{\texttt{params},P_s}$	$\mathcal{Z}$
$\mathcal{Z}$	$\longleftarrow$	P_u	$\longleftarrow$	$\mathcal{F}$				
$\mathcal{Z}$	$\xrightarrow{userid,pw}$	P_u	$\longrightarrow$	$\mathcal{F}$	$\xrightarrow{\texttt{initiate-req},userid,P_c}$	$\mathcal{S}$		
					$\xleftarrow{userid',P_c'}$			
				if $\langle userid', P_c'\rangle \neq \langle userid, P_c\rangle$ set `deflected` = true				
				$\mathcal{F}$	$\xrightarrow{\texttt{initiate-req},userid',P_c'}$	P_s	$\longrightarrow$	$\mathcal{Z}$
				$\mathcal{F}$	$\longleftarrow$	P_s	$\xleftarrow{pw'}$	$\mathcal{Z}$
				if not `deflected` and $pw == pw'$ set `auth` = valid				
				$\mathcal{F}$	$\xrightarrow{\texttt{initiate-oauth}}$	$\mathcal{S}$		
				$\mathcal{F}$	$\xleftarrow{\texttt{IssueKey2Cons, k}}$	$\mathcal{S}$		
				if `auth` = valid $AccessToken \leftarrow \$$				
$\mathcal{Z}$	$\xleftarrow{AccessToken}$	P_c	$\xleftarrow{AccessToken}$	$\mathcal{F}$				
				$\mathcal{F}$	$\xleftarrow{\texttt{IssueKey2Prov, k}}$	$\mathcal{S}$		
				if `auth` = valid				
$\mathcal{Z}$	$\xleftarrow{AccessToken}$	P_s	$\xleftarrow{AccessToken}$	$\mathcal{F}$				
$\mathcal{Z}$	$\xrightarrow{\texttt{Send},m}$	P_c	$\longrightarrow$	$\mathcal{F}$				
				if `auth` = valid	$\xrightarrow{\texttt{Send},P_c,\lvert m\rvert}$	$\mathcal{S}$		
					$\xleftarrow{+ve}$			
				$\mathcal{F}$	$\xrightarrow{\texttt{Received},m}$	P_u	$\xrightarrow{\texttt{Received},m}$	$\mathcal{Z}$

Figure 18: Example flow of the Ideal World Implementation

Implementation of $\mathcal{F}_{\text{OAUTH}^*}$ using $\mathcal{F}_{\text{SSL}}$

An example flow of the Real World Realization of $\mathcal{F}_{\text{OAUTH}^*}$ using $\mathcal{F}_{\text{SSL}}$ and $\mathcal{F}_{\text{SC}}$, with Adversary mostly responding positively.

$\mathcal{Z}$	$\xrightarrow{\texttt{initiate},P_u,P_c}$	P_u	$\xrightarrow{\texttt{Estab-sess},P_u,P_c}$	$\mathcal{F}_{\text{SSL}}^{\text{UC1}}$	$\longrightarrow$ $\xleftarrow[+ve]{}$	$\mathcal{A}$		
					$\xrightarrow{\texttt{Estab-req}}$	P_c		
		$\mathcal{A}$	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UC1}}$	$\xleftarrow{\texttt{Estab-sess}}$	P_c		
		P_u	$\xrightarrow{\texttt{Send},\langle\texttt{par-req}\rangle}$	$\mathcal{F}_{\text{SSL}}^{\text{UC1}}$	$\longrightarrow$	P_c	$\xrightarrow{\texttt{params-req}}$	$\mathcal{Z}$
$\mathcal{Z}$	$\xleftarrow{P_s}$	P_u	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UC1}}$	$\xleftarrow{\texttt{Send}}$	P_c	$\xleftarrow{\texttt{params},P_s}$	$\mathcal{Z}$
$\mathcal{Z}$	$\xrightarrow{\texttt{cred},userid,pw}$	P_u	$\xrightarrow{\texttt{Estab-sess},P_u,P_s}$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\longrightarrow$ $\xleftarrow[+ve]{}$	$\mathcal{A}$		
					$\xrightarrow{\texttt{Estab-req}}$	P_s	$\longrightarrow$	$\mathcal{Z}$
		$\mathcal{A}$	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\longleftarrow$	P_s	$\xleftarrow{\texttt{Estab-sess}}$	$\mathcal{Z}$
		P_u	$\xrightarrow{\texttt{Send},\langle userid,pw,P_c\rangle}$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\longrightarrow$	P_s	$\xrightarrow{userid,P_c}$	$\mathcal{Z}$
						P_s	$\xleftarrow{\texttt{password},pw'}$	$\mathcal{Z}$
						if $pw == pw'$		
		P_u	$\xleftarrow{Authcode}$	$\mathcal{F}_{\text{SSL}}^{\text{UP}}$	$\xleftarrow{\texttt{Send},Authcode}$	$AuthCode \leftarrow \$$		
		P_u	$\xrightarrow{\texttt{Estab-sess},P_u,P_c}$	$\mathcal{F}_{\text{SSL}}^{\text{UC2}}$	$\longrightarrow$ $\xleftarrow[+ve]{}$	$\mathcal{A}$		
					$\xrightarrow{\texttt{Estab-req}}$	P_c	$\longrightarrow$	$\mathcal{Z}$
		$\mathcal{A}$	$\longleftarrow$	$\mathcal{F}_{\text{SSL}}^{\text{UC2}}$	$\longleftarrow$	P_c	$\xleftarrow{\texttt{Estab-sess}}$	$\mathcal{Z}$
		P_u	$\xrightarrow{\texttt{Send},AuthCode}$	$\mathcal{F}_{\text{SSL}}^{\text{UC2}}$	$\longrightarrow$	P_c		
		P_c	$\xrightarrow{\texttt{Estab-sess},P_c,P_s}$	$\mathcal{F}_{\text{SC}}$	$\xleftarrow{\texttt{Estab-sess},P_s,P_c}$	P_s		
		P_c	$\xrightarrow{\texttt{Send},Authcode}$	$\mathcal{F}_{\text{SC}}$	$\xrightarrow{Authcode}$	P_s		
						if $Authcode$ same		
$\mathcal{Z}$	$\xleftarrow{AccessToken}$	P_c	$\xleftarrow{AccessToken}$	$\mathcal{F}_{\text{SC}}$	$\xleftarrow{\texttt{Send},AccessToken}$	$AccessToken \leftarrow \$$	$\xrightarrow{AccessToken}$	$\mathcal{Z}$
$\mathcal{Z}$	$\xrightarrow{\texttt{Send},m}$	P_c						
if AccessToken $\neq \perp$			$\xrightarrow{\texttt{Send},m}$	$\mathcal{F}_{\text{SSL}}^{\text{UC2}}$	$\xrightarrow{\texttt{Received},m}$	P_u	$\xrightarrow{\texttt{Received},m}$	$\mathcal{Z}$

Figure 19: Example flow of the Real World Realization of $\mathcal{F}_{\text{OAUTH}^*}$

10 Proof Automation

One of the goals of the Montage project was to make it easy for software designers and developers to adopt the framework. As we have seen in the previous chapters the key task in using the UC framework is the proof of equivalence that the real world correctly realizes the specification in the ideal world. In typical proofs of security in the UC framework we observe - firstly, it is tedious and often non-trivial to see that all possible scenarios in real execution have been analyzed; secondly, it is mostly straightforward to analyze an individual scenario, but tedious to write down. Therefore there is a strong motivation for investigating automation in the UC framework arising out of both necessity and feasibility.

This section describes a very ambitious effort to study the automation of proofs of security in the UC framework. To the best of our knowledge, ours is the first work to attempt this. The general problem is undecidable in the strong sense as it is the same as deciding if two programs are equivalent. What we show is that there are many meaningful restrictions of the problem which are decidable and further, these subsets are sufficient to model many real cryptographic protocols. While we are still very far from automating systems which are as complex as the filesystem or even the OAuth protocols, this is a very promising set of initial results.

10.1 Problem Statement

To make this section self-contained we will recall some of the basic UC terminology as well as what an application of the UC framework entails in order to motivate the automation problem. The reaader who is already knowledgeable about the details of the UC framework can directly skip to Section 10.1.2.

10.1.1 Universal Composability

The Universal Composability (UC) framework [Can] is a formal system for proving security of computational systems such as cryptographic protocols. The framework describes two probabilistic games: The *real world* that captures the protocol flows and the capabilities of an attacker, and the *ideal world* that captures what we think of as a secure system. The notion of security asserts that these two worlds are essentially equivalent.

The real-world model. The players in the real-world model are all the entities of interest in the system (e.g., the nodes in a network, the processes in a software system, etc.), as well as *the adversary* A and *the environment* $\mathcal{Z}$. All these players are modeled as efficient, probabilistic, message-driven programs (formally, they are all interactive Turing machines).

The actions in this game should capture all the interfaces that the various participants can utilize in an actual deployment of this component in the real world. In particular, the capabilities of A should capture all the interfaces that a real-life attacker can utilize in an attack on the system. (For example, A can typically see and modify network traffic.)

The environment $\mathcal{Z}$ is responsible for providing all the inputs to the players and getting all the outputs back from them. Also, $\mathcal{Z}$ is in general allowed to communicate with the adversary A. (This captures potential interactions where higher-level protocols may leak things to the adversary, etc.)

The ideal-world model. Security in the UC framework is specified via an "ideal functionality" (usually denoted $\mathcal{F}$), which is thought of as a piece of code to be run by a completely trusted entity in the ideal world. The specification of $\mathcal{F}$ codifies the security properties of the component at hand. Formally, the ideal-world model has the same environment as the real-world model, but we pretend that there is a completely trusted party (called "the functionality"), which is performing all the tasks that are required of the protocol. In the ideal world, participants just give their inputs to the functionality $\mathcal{F}$, which produces the correct outputs (based on the specification) and hands them back to the participants. $\mathcal{F}$ may interact with an adversary, but only to the extent that the intended security allows. (E.g., it can "leak" to the adversary things that should be publicly available, such as public keys.)

UC-Security. An implementation $\mathcal{P}$ **securely realizes** an ideal functionality $\mathcal{F}$ if no external environment can distinguish between running the protocol $\mathcal{P}$ in the real world and interacting with the trusted entity running the ideal functionality $\mathcal{F}$ in the ideal world. That is, for every adversary A in the real world, there should exist an adversary A' in the ideal world, such that no environment $\mathcal{Z}$ can distinguish between interacting with A and $\mathcal{P}$ in the real world and interacting with A' and $\mathcal{F}$ in the ideal world. A remarkable feature of UC-Security is that the security guarantees are preserved under modular compositions.

With this background we are ready to state the proof automation problem.

10.1.2 Proof Automation Problem

As discussed, the real protocol is a distributed system which can be modeled as a set of algorithms $\mathcal{P}_1, \mathcal{P}_2, \cdots$ running concurrently at various locations in the network. Typically these are roles like clients, servers and peers running code according to the rules of the protocol. The messages they exchange are in the open and can be arbitrarily blocked, modified, rerouted or mixed by the adversary. To preserve integrity and confidentiality in the insecure channel, the parties may use cryptographic primitives like signatures and encryptions. Each party may run one or more instance of the roles (algorithms $\mathcal{P}_i$) concurrently. A party may be client in one session and server in another.

The ideal functionality is also a set of algorithms $\mathcal{F}_1, \mathcal{F}_2, \cdots$ which abstract the oracular computations of the ideal trusted entity. This is an abstract specification of the security of the system. For example, secure channels may be abstracted away by the functionalities themselves transmitting messages between parties, without the adversary knowing anything about the message, except perhaps the length. Multi-party computations can be abstracted by all parties submitting their individual arguments to a central functionality ("a trusted

third party"), and the central functionality then computing the function over the arguments and transmitting the result to all the parties.

A proof of security is to show that all possible adversarial behavior with the real protocol $\mathcal{P}$ can be efficiently translated to an adversarial behavior with the functionality $\mathcal{F}$, thus effectively showing that if the functionilty captures the security requirements, so does the protocol.

Therefore, formally, a proof of security in the UC model boils down to the following:

As input, we are given a set of principals and two sets of algorithms:

1. Real Protocol: Set of algorithms $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \cdots\}$.
2. Ideal Functionality: Set of algorithms $\mathcal{F} = \{\mathcal{F}_1, \mathcal{F}_2, \cdots\}$.

We say that $\mathcal{P}$ realizes $\mathcal{F}$ if it is possible to construct an algorithm S, called a simulator, that invokes the functions in $\mathcal{F}$, such that the following holds: For any sequence of calls to algorithms in $\mathcal{P}$, S can come up with a sequence of calls such that the "effect" is "same".

Note that it is not sufficient to construct a simulator for every fixed call sequence to the $\mathcal{P}_i$'s. This is because the adversary may be adaptive and decide to call a certain $\mathcal{P}_i$ based on the messages it has seen so far and the random coins it is using. Hence, the simulator has to be adaptive too. Technically, for any prefix of a call sequence, the simulator should be able to come up with a strategy such that it can match the adversary no matter which path it takes after the prefix. In this article, we focus attention on constructing simulators for a fixed call sequence only. How to simulate adaptive adversaries remains an open problem.

The words "effect" and "same" have a technical explanation which we will get to in the subsequent section, but here the reader can just think of them as feasibly observable properties. An example "effect" can be an output quantity at the end and "same" could be identical value or close enough probability distribution. The standard model of protocol execution, captured in [Can], consists of a set of distributed algorithms representing the parties running the protocol, plus an algorithm representing the adversary. The adversary controls a subset of the parties, which in general may be chosen adaptively throughout the execution. In addition, the adversary has some control over the scheduling of message delivery. The parties and adversary interact on a given set of inputs and each party eventually generates local output. The concatenation of the local outputs of the adversary and all parties is called the global output. In the ideal process for evaluating some function f all parties ideally hand their inputs to an incorruptible trusted party, who computes the function values and hands them to the parties as specified. Here the adversary is limited to interacting with the trusted party in the name of the corrupted parties. Protocol $\mathcal{P}$ securely evaluates a function f if for any adversary A (that interacts with the protocol) there exists an ideal-process adversary S such that, for any set of inputs to the parties, the global output of running $\mathcal{P}$ with A is indistinguishable from the global output of the ideal process for f with

adversary S.

The proof automation problem is to come up with the simulator algorithm S given $\mathcal{F}$ and $\mathcal{P}$, or show that none exists. Consider a fixed call sequence of $\mathcal{P}$. The strategy we develop is the following:

1. Finiteness Argument: We show that there is a bounded set of finite call sequences of $\mathcal{F}$ that captures all possible effects of unbounded call sequences.
2. Theorem Proving: We enumerate all such call sequences and check for equivalence with the given call sequence of $\mathcal{P}$.
3. Output: If there is a match, we output the simulator. If none of the elements of this bounded set matches, then no simulation is possible.

Of course, the Finiteness Argument and Equivalence Checking are only possible only in restricted program constructs - otherwise the problems become equivalent to known undecidable problems: If the $\mathcal{F}$'s were allowed to do operations like simulate any turing machine's tape operations, all possible effects cannot be obviously be captured by bounded call sequences to $\mathcal{F}$. Moreover, given a simulator description and the protocol description, Equivalence checking is the same or harder than general program equivalence checking, which is known to be undecidable.

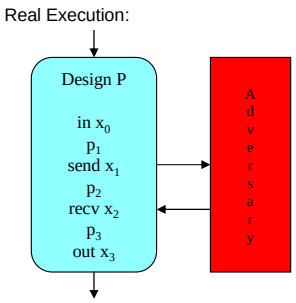
The technical challenge is to come up with program classes that have the property required for a finiteness argument. As we show in the subsequent sections, even with a fair degree of restrictions we step into undecidability. However, we also come up with interesting language classes, motivated by cryptography, which have the finiteness property.

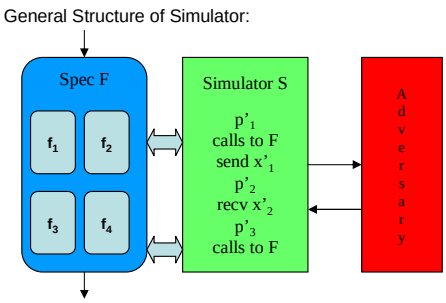
We sketch a general decision procedure in Figure 21, with the high level functions explained shortly. The components that go into the correctness and operation of this procedure are as follows:

10.1.2.1 Finiteness Argument. This component is a mathematical proof that a bounded set of finite call sequence exists for the given language class. This needs to be done once per language class.

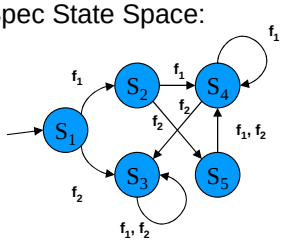
10.1.2.2 Call Sequence Enumeration. This component is an algorithm that given $\mathcal{F}$, generates call sequences that include all the call sequences in the bounded set for the given $\mathcal{F}$.

10.1.2.3 Equivalence Checking. This component is an algorithm which checks whether a given call sequence matches the given protocol serialization.





Spec State Space:



- Capture the equivalence of the communication behavior of the players.
- In between the adversary/environment interactions, the simulator can call the component "spec" arbitrarily often and in any order
- Prove that for the given language, the "effect" of an arbitrary call sequences can be modeled by a call sequence belonging to a finite set.

1

Figure 20: Proof automation: technique

Algorithm GenerateSimulator$(\mathcal{F}, \mathcal{P})$

```
Repeat ⟨F_1, F_2, ··· , F_k⟩ ← CallSequenceEnumerator(F)
    SimulatorDescription ← ∅
    Repeat in sequence F_i ← ⟨F_1, F_2, ··· , F_k⟩
        Arguments ← GetArguments(SimulatorDescription)
        t ← F_i(Arguments)
        Add t to SimulatorDescription
    Until end of sequence
    Call Theorem Prover to check if SimulatorDescription matches P.
    If so, output SimulatorDescription and halt
    Else continue
Until call sequence enumeration is complete
Output "No simulator exists."
```

Figure 21: Structure of a general decision procedure

For each functionality $\mathcal{F}_i$ in the call sequence, computed iteratively by the function *CallSequenceEnumerator* in Figure 21, the simulator computes arguments for the invocation of $\mathcal{F}_i$ and stores the output obtained. The complexity in mathematically analyzing the call sequence structures arise from the following: firstly, each invocation may build up state in the local storages of the functionalities, which may be persistent across calls. Secondly, computing arguments (the *GetArguments* function in Figure 21) for the next function invocation has to be designed carefully.

The Theorem Prover takes a given simulator description (*SimulatorDescription* in Figure 21), and checks for equivalence against $\mathcal{P}$. This consists of checking symbolically whether interaction and exchange of messages with the environment and the adversary are equivalent for all possible inputs from the environment and adversary.

10.1.3 Motivating Example.

Password-based key exchange is an important security problem which has been studied extensively in cryptographic research [BM93], and which brings out the power of the UC framework particularly well. Canetti et al [CHK+05] proposed an Ideal Functionality for password-based key exchange which is formally described in Figure 22.

Consider two parties P_i and P_j that wish to come up with a common cryptographically

Functionality $\mathcal{F}_{\text{pwKE}}$

The functionality $\mathcal{F}_{\text{pwKE}}$ is parameterized by a security parameter k. It interacts with an adversary S and a set of parties via the following queries:

Upon receiving a query $(\mathsf{NewSession}, sid, P_i, P_j, pw, role)$ **from party** P_i**:**
Send $(\mathsf{NewSession}, sid, P_i, P_j, role)$ to S. In addition, if this is the first NewSession query, or if this is the second NewSession query and there is a record (P_j, P_i, pw'), then record (P_i, P_j, pw) and mark this record fresh.

Upon receiving a query $(\mathsf{TestPwd}, sid, P_i, pw')$ **from the adversary** S**:**
If there is a record of the form (P_i, P_j, pw) which is fresh, then do: If $pw = pw'$, mark the record compromised and reply to S with "correct guess". If $pw \neq pw'$, mark the record interrupted and reply with "wrong guess".

Upon receiving a query $(\mathsf{NewKey}, sid, P_i, sk)$ **from** S**, where** $|sk| = k$**:**
If there is a record of the form (P_i, P_j, pw), and this is the first NewKey query for P_i, then:

- If this record is compromised, or either P_i or P_j is corrupted, then output (sid, sk) to player P_i.
- If this record is fresh, and there is a record (P_j, P_i, pw') with $pw' = pw$, and a key sk' was sent to P_j, and (P_j, P_i, pw) was fresh at the time, then output (sid, sk') to P_i.
- In any other case, pick a new random key sk' of length k and send (sid, sk') to P_i.

Either way, mark the record (P_i, P_j, pw) as completed.

Figure 22: The password-based key-exchange functionality $\mathcal{F}_{\text{pwKE}}$

strong key based on the fact that they share the same password. The idea is to capture the fact that modulo the adversary outright guessing the password exactly during an active session between the parties, it has no control (or information) on the key being generated. It is allowed to interrupt sessions by tampering with the messages being exchanged, but doing so only results in the parties ending up with different uniformly randomly distributed keys. If, however, the session is not interrupted, the parties end up with the same key which is distributed uniformly and randomly and is not controlled by the adversary.

In the following discussion, we will overlook the session ids for simplicity although it is straightforward to add them. We describe a protocol Π_{ICpwKE}, in Figure 23, which is a candidate to realize $\mathcal{F}_{\text{pwKE}}$. This is a protocol based on the Ideal Cipher model [BPR00]. In the ideal cipher model, the results of two decryptions are the same if the key is identical. Otherwise, the results are uniformly and independently random. The protocol is symmetric

from the perspective of both the participants - so we describe the actions of just one party P_i. Both parties get a password from the environment $\mathcal{E}$. Party P_i generates a random number r_1, encrypts it and sends the ciphertext c_1 to the peer. When it receives a response c_2', it first checks whether its own message was reflected. If so, it outputs a random key to the environment. Otherwise, it decrypts the response using its password pw_1 and xors the plaintext with r_1. The resulting quantity is output as its key to the environment.

Party P_i	Adv	Party P_j
$\mathcal{E}$		$\mathcal{E}$
$\downarrow pw_1$		$\downarrow pw_2$
$r_1 \leftarrow \$$		$r_2 \leftarrow \$$
$c_1 \leftarrow enc_{pw_1}(r_1)$		$c_2 \leftarrow enc_{pw_2}(r_2)$
	$\xrightarrow{c_1}$ $\xleftarrow{c_2}$	
	$\xleftarrow{c_2'}$ $\xrightarrow{c_1'}$	
`if` $(c_2' == c_1)$ `then` $sk_1 \leftarrow \$$		`if` $(c_1' == c_2)$ `then` $sk_2 \leftarrow \$$
`else`		`else`
$d_1 \leftarrow dec_{pw_1}(c_2')$		$d_2 \leftarrow dec_{pw_2}(c_1')$
$sk_1 \leftarrow r_1 \oplus d_1$		$sk_2 \leftarrow r_2 \oplus d_2$
$\downarrow sk_1$		$\downarrow sk_2$
$\mathcal{E}$		$\mathcal{E}$

Figure 23: Protocol for Password-based Key Exchange using Ideal Cipher.

Consider the following *ideal functionality for the ideal cipher* primitive. The functionality takes two arguments: a key and a plaintext. It has a table where each entry is a triplet (key, plaintext, ciphertext). The table is initially empty. It supports two subroutines: encrypt(key, plaintext) and decrypt(key, ciphertext). The **encrypt** subroutine, given input $(key, plaintext)$, generates a random number r, stores $(key, plaintext, r)$ in the table and outputs r. The **decrypt** subroutine, given input $(key, ciphertext)$, looks up if there is an entry $(key, p, ciphertext)$ in the table. If so, it outputs p. Otherwise, it generates a random number r, stores $(key, r, ciphertext)$ in the table and outputs r.

Now consider the real-world scenario where the adversary intercepts the first message c_1 and changes it to c_1' before transmitting to P_j. The adversary's action may involve querying

the ideal cipher in the hybrid model. More importantly, if the password is weak, the adversary maybe able to guess the password, and hence a proper simulation would require the simulator to extract this password guess from the call to the ideal cipher, and use that in the TestPwd subroutine of $\mathcal{F}_{\text{pwKE}}$.

We will revisit this example in Section 10.5.2 in more technical details when we have the necessary context. In the current context, we instantiate the generic steps of our technique that we outlined.

Finiteness Argument. We demonstrate that for the operators used in the description of the protocol and the functionality, that is, equality testing, conditional branches, xors, random number generation and bounded storage, the number of possible simulators is finite.

Call Sequence Enumeration. These simulators use a sequence of a priori described operations. Therefore, the way to generate a simulator is to generate all possible sequences within a fixed bound derived from the finiteness argument step.

Equivalence Checking. In essence, the simulators are special multivariate polynomials on finite fields of characteristic two. We show that these special polynomials can be canonically described over a well behaved basis. Thus to check equivalence of the simulator with the real protocol execution, we just check if they split equivalently over the basis.

10.2 Overview of Results

The language classes we investigated are inspired by formulations of standard cryptographic primitives as we will exemplify in Section 10.5.2. In exploring various choices of languages, we came up with both negative and positive results:

Negative results: We begin with the following observation:

Observation 19 *If both the ideal and real systems are allowed to be arbitrary Turing Machines, then the problem becomes as hard as deciding program equivalence, which is undecidable.*

Therefore we have to restrict the class of systems that we allow in order to come up with decision procedures. To formalize the restrictions, we fix a language to describe the systems. The language is restricted in the sense that one cannot frame arbitrary turing computations in them, but at the same time it allows specification of programs motivated by cryptographic protocols.

We showed that even under fairly restricted scenarios, *the problem remains undecidable.* In particular, this arises when unbounded table lookup/storage operations are allowed (not even random access, but just storing and detecting whether some string is there in the table), even when there are no arithmetic operations, loops in subroutines or random number generation. Formally, we have the following theorem:

Theorem 5 (An Undecidable System) *Let L be a language with input / outputs to the environment, send / receives to the adversary, conditional with equality checking of strings and table storage/lookup. We are given a real protocol $\mathcal{P}$ and a ideal functionality $\mathcal{F}$ with all subroutines described in L. We show that it is impossible to algorithmically decide whether $\mathcal{P}$ realizes $\mathcal{F}$.*

Positive results: We restricted the language further to avoid the undecidable territories and came up with a decision procedure for a language that allows environment input / outputs, adversary send / receives, assignment statements, xor operations, random number generation and finite local storage. We will see in Section 10.5.2 how these operators are used in specification of protocols and functionalities. Formally, we have the following result:

Theorem 6 (A Decidable System) *Let L be a language with input / outputs to the environment, send / receives to the adversary, assignment statements, xor operations, random number generation, conditional statements, constant number of storage elements and uninterpreted functions. We are given a real protocol $\mathcal{P}$ and a ideal functionality $\mathcal{F}$ with all subroutines described in L. We develop a decision procedure to find whether $\mathcal{P}$ realizes $\mathcal{F}$. We restrict $\mathcal{P}$ to be a single program, whereas $\mathcal{F}$ can have multiple subroutines which can be called an unbounded number of times in any arbitrary order.*

10.3 Restriction to Language Classes

The language classes we look at have operators motivated by UC models of cryptographic primitives. As an example, we reproduce below the description of a Universally Composable Commitment Scheme from [CF01] which realizes the Ideal Commitment Functionality. Commitment is one of the most basic and useful cryptographic primitives. On top of being intriguing by itself, it is an essential building block in many cryptographic protocols, such as Zero-Knowledge protocols, general function evaluation protocols, contract-signing and electronic commerce, and more. The basic idea behind the notion of commitment is attractively simple: A committer provides a receiver with the digital equivalent of a "sealed envelope" containing a value x. From this point on, the committer cannot change the value inside the envelope, and, as long as the committer does not assist the receiver in opening the envelope, the receiver learns nothing about x. When both parties cooperate, the value x is retrieved in full.

The Ideal Commitment functionality is described as follows:

Functionality F_{com}:

F_{com} proceeds as follows, running with parties $P_1, \cdots, P_n$ and an adversary S.

1. Upon receiving a value $(\texttt{Commit}, sid, P_i, P_j, b)$ from P_i, where $b \in \{0,1\}$, record the value b and send the message $(\texttt{Receipt}, sid, P_i, P_j)$ to P_j and S. Ignore any subsequent Commit messages.

2. Upon receiving a value $(\texttt{Open}, sid, P_i, P_j)$ from P_i, proceed as follows: If some value b was previously recorded, then send the message $(\texttt{Open}, sid, P_i, P_j, b)$ to P_j and S and halt. Otherwise halt.

The commitment phase is modeled by having F_{com} receive a value $(\texttt{Commit}, sid, P_i, P_j, b)$, from some party P_i (the committer). Here sid is a Session ID used to distinguish among various copies of F_{com}, P_j is the identity of another party (the receiver), and $b \in \{0,1\}$ is the value committed to. In response, F_{com} lets the receiver P_j and the adversary S know that P_i has committed to some value, and that this value is associated with session ID sid. This is done by sending the message $(\texttt{Receipt}, sid, P_i, P_j)$ to P_j and S. The opening phase is initiated by the committer sending a value $(\texttt{Open}, sid, P_i, P_j)$ to F_{com}. In response, F_{com} hands the value $(\texttt{Open}, sid, P_i, P_j, b)$ to P_j and S.

The protocol $\text{UCC}_{OneTime}$, which is claimed to realize F_{com}, is described below:

Commitment Scheme $\text{UCC}_{OneTime}$:

public string:

$\sigma \leftarrow$ random string in $\{0,1\}^{4n}$

$pk_0, pk_1 \leftarrow$ keys for generators $G_{pk_0}, G_{pk_1} : \{0,1\}^n \rightarrow \{0,1\}^{4n}$

commitment for $b \in \{0,1\}$ with SID sid:

compute $G_{pk_b}(r)$ for random $r \in \{0,1\}^n$

set $y = G_{pk_b}(r)$ for $b = 0$, or $y = G_{pk_b}(r) \oplus \sigma$ for $b = 1$

send $(\texttt{Com}, sid, y)$ to the receiver

Upon receiving $(\texttt{Com}, sid, y)$ from P_i, P_j outputs $(\texttt{Receipt}, sid, cid, P_i, P_j)$

decommitment for y:

send b, r to the receiver

receiver checks $y == G_{pk_b}(r)$ for $b = 0$, or $y == G_{pk_b}(r) \oplus \sigma$ for $b = 1$.

If the verification succeeds then P_j outputs $(\texttt{Open}, sid, P_i, P_j, b)$.

Let $KGen$ denote an efficient algorithm that on input 1^n generates a random public key pk and the trapdoor td. G_{pk} is a pseudorandom generator derived from pk. An important feature of this generator is that given the trapdoor td to pk it is easy to tell whether a given $y \in \{0,1\}^{4n}$ is in the range of G_{pk}. The public random string in our scheme consists of a random $4n$-bit string σ, together with two public keys pk_0, pk_1 describing trapdoor pseudorandom generators G_{pk_0} and G_{pk_1}; both generators stretch n-bit inputs to $4n$-bit output. The public keys pk_0, pk_1 are generated by two independent executions of the key generation algorithm $KGen$ on input 1^n.

The proof of realization of F_{com} by $\text{UCC}_{OneTime}$ is by constructing a simulator which simulates the actions of $\text{UCC}_{OneTime}$ by just invoking F_{com}, such that no environment can tell whether it is interacting with a real instance of $\text{UCC}_{OneTime}$ or the simulator invoking F_{com}.

As we can see, the following operators are used in describing the protocols:

1. Inputs from the environment and outputs to the environment.
2. Send and receive messages with the adversary.
3. Random number generation.
4. Assignment operations.
5. Taking XOR of two bitstrings.
6. Conditional execution.

Public-key encryption and signature are two important primitives which are modeled using table storage and lookups. The following describes the Ideal Functionality for signature schemes [Can]:

Functionality F_{sig}:

- Key Generation: Upon receiving a value $(\texttt{KeyGen}, sid)$ from some party S, verify that $sid = (S, sid')$ for some sid'. If not, then ignore the request. Else, hand $(\texttt{KeyGen}, sid)$ to the adversary. Upon receiving $(\texttt{Algorithms}, sid, s, v)$ from the adversary, where s and v are descriptions of feasible algorithms, output $(\texttt{Verification Algorithm}, sid, v)$ to S.
- Signature Generation: Upon receiving a value $(\texttt{Sign}, sid, m)$ from S, let $\sigma = s(m)$, and verify that $v(m, s) = 1$. If so, then output $(\texttt{Signature}, sid, m, \sigma)$ to S and record the entry (m, σ). Else, output an error message to S and halt.
- Signature Verification: Upon receiving a value $(\texttt{Verify}, sid, m, \sigma, v')$ from some party V, do: If $v' = v, v(m, \sigma) = 1$, and no entry (m, σ') for any σ' is recorded, then output an error message to S and halt. Else, output $(\texttt{Verified}, sid, m, v'(m, \sigma))$ to V.

The basic idea of F_{sig} is to provide a "registry service" where the signer S can register ($message$, $signature$) pairs. Any party that provides the right verification key can check whether a given pair is registered. F_{sig} takes three types of inputs, which correspond to the three basic modules of a signature scheme: key generation, signature generation, and signature verification. Having received a key generation request from a party S, F_{sig} first verifies that the identity S appears in the SID. This convention essentially guarantees that each party can invoke F_{sig} with an SID that no other party can use. Next, F_{sig} asks the adversary to provide two descriptions of algorithms: A polynomial-time probabilistic signing algorithm s, and a polynomial-time deterministic verification algorithm v. It then outputs to S the description of the signing algorithm s. Finally, it invokes algorithms s and v, and maintains a running instance of each.

Upon receiving a request from party S (and only party S) to sign a message m, F_{sig} first obtains a "formal signature string" by running s on input m and fresh random input. It then verifies that $v(m, \sigma) = 1$. If so, it outputs the signature string σ to S and records the pair (m, σ). Else, F_{sig} outputs an error message and halts.

Upon receiving a request from some arbitrary party V to verify a signature σ on message m with verification algorithm (or, key) v', F_{sig} proceeds as follows. It first checks if the input consists of a forgery, namely if $v' = v, v(m, \sigma) = 1$, and S is uncorrupted and never signed m in the past (namely, the pair (m, σ') is not recorded, for any σ'). If so, F_{sig} outputs an error message and halts. Else, it outputs $v'(m, \sigma)$.

The fact that the verification algorithm v is deterministic guarantees *consistency*, namely that all verification requests for the same triple (m, σ, v') are answered in the same way, even if made by different parties at different times. Verifying that $v(m, \sigma) = 1$ at signature generation time guarantees *completeness*, namely that if a signature was generated "honestly" (i.e., via F_{sig}) then it will verify. *Unforgeability* is guaranteed by the check for forgery at verification time.

10.4 Decision Procedures

In this section, we develop decision procedures for a specific language which includes xor, conditional and random number generation operations over finite fields of characteristic two. To describe the semantics of this language, we consider multivariate pseudo-linear functions, which are functions computed by branching programs over data objects from additive group of fields of characteristic two. The conditionals in such programs are built from equality constraints over linear expressions, closed under negation and conjunction.

Let $f_1, f_2, ..., f_k$ be k pseudo-linear functions in n variables, and let f be another pseudo-linear function in the n variables. We show that if f is a function of the given k functions, then it must be a pseudo-linear function of the given k functions. This generalizes the straightforward claim for just linear functions. We also prove a more general theorem where the k functions can in addition take further arguments, and prove that if f can be represented as an iterated composition of these k functions, then it can be represented as a probabilistic pseudo-linear iterated composition of these functions. Proceeding further, we generalize

the theorem to randomized pseudo-linear functions. Additionally, we allow f itself to be a randomized function, i.e. we give a procedure for deciding if f is a probabilistic sub-exponential in m time iterated function of the given k randomized functions, and the decision procedure runs in computational time independent of m.

These theorems have implications for automatic proving of universally-composable security theorems for ideal and real functionalities composed of if-then-else programs with (uniform) random number generation and data objects from additive group of GF(2^m). The theorems imply that, within this language framework, there is a decision procedure to find out if a real functionality realizes an ideal functionality, and this procedure is in computational time independent of m (which is essentially the security parameter).

10.4.1 Pseudo-Linear Functions

Before we define pseudo-linear functions, we mention that pseudo-linear functions originate as functions computed by if-then-else or branching programs involving data objects from the additive group of fields of characteristic two. The conditionals are built from equality constraints of linear expressions, and closed under negation and conjunction.

So, consider a finite field $\mathcal{F}_q$, where $q = 2^m$. Then m-bit (bit-wise) exclusive-OR just corresponds to addition in this field. Further, an equality constraint of the form $l_1(\vec{x}) = l_2(\vec{x})$ can then be written as $1 + (l_1(\vec{x}) + l_2(\vec{x}))^{q-1}$ which evaluates to 1 if $l_1(\vec{x}) = l_2(\vec{x})$, and evaluates to zero otherwise. Similarly, $l_1(\vec{x}) = 0$ *and* $l_2(\vec{x}) = 0$ can be written as $(1 + l_1(\vec{x})^{q-1}) \cdot (1 + l_2(\vec{x})^{q-1})$ As a final example, an expression "if ($l_1(\vec{x}) = 0$ *and* $l_2(\vec{x}) = 0$) then $l_3(\vec{x})$ else $l_4(\vec{x})$" can be written as $(1 + l_1(\vec{x})^{q-1}) \cdot (1 + l_2(\vec{x})^{q-1}) \cdot (l_3(\vec{x}) + l_4(\vec{x})) + l_4(\vec{x})$

A **pseudo-linear** multivariate polynomial defined over sub-field $\mathcal{F}_2$ is then a polynomial which is a sum of guarded linear-terms [Dij75]; a *guarded linear-term* is a polynomial which is the product of a linear (over $\mathcal{F}_2$) polynomial and zero or more linear-guards; a *linear-guard* is a linear (over $\mathcal{F}_2$) polynomial raised to the power q-1. Since, in this section we will only be dealing with pseudo-linear polynomials defined over $\mathcal{F}_2$, from now on we will implicitly assume that. A pseudo-linear polynomial in n variables and defined over $\mathcal{F}_2$, however does yield a function from $(\mathcal{F}_q)^n$ to $\mathcal{F}_q$, which we call a **pseudo-linear function**. Thus, even though the polynomial is defined over $\mathcal{F}_2$, the underlying field will be $\mathcal{F}_q$, and hence the algebra of the polynomials is modulo $(x_i^q = x_i)$ (for i ranging from 1 to n). In formal terms, the objects in consideration are in $\mathcal{F}_2[x_1, ..., x_n]/(x_1^q + x_1, ..., x_n^q + x_n)$. They are also further restricted by the fact that all expressions in the guards are linear instead of affine, but we will later see how to introduce constant additive terms from $\mathcal{F}_q$ (Section 10.4.2.3).

We observe that pseudo-linear polynomials are closed under pseudo-linear transformations, i.e. given a pseudo-linear polynomial, raising it to the power q-1, and multiplying it by another pseudo-linear polynomial yields just another pseudo-linear polynomial. This follows (by induction) from the observation that

$$(f_1(\vec{x})^{q-1} l_1(\vec{x}) + f_2(\vec{x}))^{q-1} = f_1(\vec{x})^{q-1}(l_1(\vec{x}) + f_2(\vec{x}))^{q-1} + (1 + f_1(\vec{x})^{q-1}) \cdot f_2(\vec{x})^{q-1} \quad (2)$$

where f_1 and f_2 are arbitrary pseudo-linear functions. The observation itself follows by considering the two cases where $f_1(\vec{x})$ is zero or not.

More importantly, the if-then-else programs mentioned above compute exactly the pseudo-linear functions. A more detailed description of such programs and how they relate to pseudo-linear functions can be found in Section 10.5.

Above, we saw how a multivariate polynomial $p(\vec{x})$ yields a function from $\mathcal{F}_q^n$ to $\mathcal{F}_q$. More generally, if we are given n polynomials $f_1(\vec{z})$ to $f_n(\vec{z})$ (where $\vec{z}$ are k formal variables), then $p(f_1(\vec{z}), ..., f_n(\vec{z}))$ yields a function from $\mathcal{F}_q^k$ to $\mathcal{F}_q$, which we say is a **pseudo-linear function** of $f_1, ..., f_n$.

While for linear multivariate functions a *completeness theorem* which states that if a linear function f of n variables is a function of k other linear functions (in the same n variables), then f must be a *linear function* of the k linear functions, is well known and rather easy to prove, a similar completeness result for pseudo-linear functions is novel and not so easy to prove.

Thus, one of the main result of this section is a theorem which states that if a pseudo-linear multivariate function f of n variables is a *function* of k pseudo-linear functions $f_1, f_1, ..., f_k$ (in the same n variables), then f must be a *pseudo-linear* function of $f_1, f_2, ..., f_k$. Note that it is given that f itself is a pseudo-linear function in the original n variables.

The primitives in this algebraic structure are motivated by cryptography, in particular the representation of security properties and cryptographic protocols in the the Universally Composable (UC) framework. The core technique involved in the UC framework is to prove equivalence of the system in consideration - the real protocol - and an ideal functionality which naturally captures the security requirements. The proof is by demonstrating a *simulator* which has access to the ideal functionality and is able to produce results indistinguishable from an execution of the target function.

To model the fact that a simulator can iteratively compose various calls to the different functions in the ideal functionality, we *prove a more general theorem* involving arbitrary iterations of k functions $f_1(\vec{z}, \vec{y}), f_2(\vec{z}, \vec{y}), ..., f_k(\vec{z}, \vec{y})$, where $\vec{y}$ are arguments which the simulator can supply. We then prove that if some f is a pseudo-linear function of $\vec{z}$, and can be computed by a sub-exponential (in m) length iterated composition of the given k functions, then it can be computed by a *probabilistic* iterated *pseudo-linear* composition of the given k functions, i.e. f_1, f_2, ..., f_k.

Proceeding further, we include random number generation as an additional primitive and extend the decision procedure to find if a *probabilistic poly time* simulator exists for the given set of *randomized* ideal functionalities and *randomized* target function.

For cryptographic applications, this means that an algorithmic search for a simulator in proving that a protocol in this language realizes an ideal functionality (also in this language) is *independent* of the security parameter, as the security parameter is usually related to the field size. Since the program sizes in cryptographic protocols are usually small, this can lead to efficient theorem proving. There are additional issues involved, e.g. the real protocol may be given in a hybrid model [Can], and the adversary may make iterative calls to the hybrid functionalities.We do not deal with computational assumptions in this section, and we expect that the hybrid functionalities themselves embody such assumptions (see e.g. [CG10]). We discuss how our work is motivated by the UC framework and give

an example in Section 10.5.2. Although, there have been many pieces of work in formal methods for cryptographic protocols [AR00, CH, MW04, DDMR07b], this to the best of our knowledge is a novel approach to theorem proving of security protocols.

There is a technical restriction of a sub-exponential length iterated composition which is required to rule out deterministic brute force searches, which a computationally bounded simulator is not allowed anyway. Finally, we remark that our completeness results require sufficiently large fields (as a function of the number of variables in f), but given that most UC proofs only seek proofs of simulatability which do not depend on the security parameters, our completeness theorem covers all such UC proofs.

The difficulty in proving the completeness theorem stems from the fact that pseudo-linear polynomials can have individual degrees (i.e. of individual variables) exceeding q-1, and hence it may be subject to reduction modulo $x^q = x$. Similar problems occur in local testing of low degree polynomials [JPRZ04, KR04], and we would like to point out that pseudo-linear functions are intimately related to Generalized Reed-Muller Codes [KLP68]. Thus, for example it is not immediately clear what constitutes a basis for pseudo-linear polynomials. We first show a basis for pseudo-linear polynomials, and then show a necessary and sufficient condition involving the basis for a pseudo-linear function of $\vec{x}$ to be a pseudo-linear function of other pseudo-linear functions of $\vec{x}$. A detailed example illustrating these issues and the proof idea can be found in Section 10.4.1.1. We would also like to mention that the class of pseudo-linear functions do not form an ideal in $\mathcal{F}_q[\vec{x}]$, and hence the vast field of Gröbner basis is not applicable.

We remark that our theorem does not yet address stateful functions. Many important functionalities, e.g Random Oracle, Public Key Encryption etc. require stateful functionalities [Can]. Moreover, these functionalities require arbitrarily large tables of state, although the entries in the table are of fixed size (i.e. depend only on m). This is important to note, as we show that with arbitrarily large sized entries in tables, the question of simulatability is undecidable (see Section 10.6). However, we expect our positive results to extend to stateful functionalities, and also to functionalities with tables with limited capabilities, e.g. fixed sized entries. We also expect our results to extend to other groups and cryptographic constructs with appropriate axiomatization.

10.4.1.1 Example of Pseudo-Linear Functions We will consider some simple examples to get a flavor of the problem. Suppose we are given two input functions f_1 and f_2 defined as follows:

$$f_1(x_1, x_2) = x_1 + x_2 \tag{3}$$

$$f_2(x_1, x_2) = \left\{ \begin{array}{ll} 0 & \text{if } x_1 = 0 \text{ or } x_2 = 0 \\ x_1 + x_2 & \text{otherwise} \end{array} \right\} = x_2^{q-1} x_1 + x_1^{q-1} x_2 \tag{4}$$

We ask if it is possible to extract just x_1 given $f_1(x_1, x_2)$ and $f_2(x_1, x_2)$. That is, can we express $f(x_1, x_2) = x_1$, in terms of f_1 and f_2 alone? To do so, we construct the following

truth table:

	x_1	x_2	$x_1 + x_2$	f_1	f_2	f
Row 1	0	0	0	0	0	0
Row 2	x_1	0	x_1	0	x_1	x_1
Row 3	0	x_2	x_2	0	x_2	0
Row 4	x_1	x_1	0	0	0	x_1
Row 5	x_1	x_2	$x_1 + x_2$	$x_1 + x_2$	$x_1 + x_2$	x_1

In the table above we list all linear combinations of the atoms, in this case just x_1, x_2 and $x_1 + x_2$. Each row corresponds to different combinations of cases where each linear combination can be zero or non-zero. Any non-zero entry under a column means that the particular linear combination is non-zero. Simplifications are performed when some of the linear expressions are zero - e.g. Row 4, where we write x_1 under the column x_2 since $x_1 + x_2 = 0 \Rightarrow x_2 = x_1$. It turns out that any pseudo-linear expression projects to a linear expression in any particular row - thus each such function can be given by a column of linear expressions, e.g. f_1, f_2 and f above. In this particular table for f_1, f_2 and f we can come up with several evidences that f is not a function of f_1, f_2. Consider Row 4: both f_1 and f_2 are 0, whereas f is x_1. Therefore, in accordance with the structure of the row, if we vary x_1, keeping it non-zero and $x_2 = x_1$, we get two pairs (x', x') and (x'', x'') with $x' \neq x''$ such that $f_1(x', x') = f_1(x'', x'') = 0$ and $f_2(x', x') = f_2(x'', x'') = 0$, but $f(x', x') = x' \neq x'' = f(x'', x'')$. Hence f cannot be a function of f_1, f_2. We can construct a counterexample using Row 5 as well: vary x_1 keeping $x_1 + x_2$ constant and keeping $x_1, x_2, x_1 + x_2$ all non-zero - e.g. in $\mathrm{GF}(2^3) : (x_1', x_2') = (001, 010)$ and $(x_1'', x_2'') = (101, 110)$. The common evidence in both rows is that f is not a linear combination of f_1, f_2.

However, this is not the only type of evidence. Consider the following $f'(x_1, x_2)$:

$$f'(x_1, x_2) = \left\{ \begin{array}{ll} x_1 & \text{if } x_2 = 0 \\ 0 & \text{otherwise} \end{array} \right\} = (1 + x_2^{q-1})x_1 \tag{5}$$

Now the table looks like:

	x_1	x_2	$x_1 + x_2$	f_1	f_2	f'
Row 1	0	0	0	0	0	0
Row 2	x_1	0	x_1	0	x_1	x_1
Row 3	0	x_2	x_2	0	x_2	0
Row 4	x_1	x_1	0	0	0	0
Row 5	x_1	x_2	$x_1 + x_2$	$x_1 + x_2$	$x_1 + x_2$	0

Now in each row, f' is a linear combination of f_1, f_2 (including the 0-combination). However, there is a problem with Rows 2 and 3. The problem surfaces when we try to write f' as a combination of f_1, f_2:

	x_1	x_2	$x_1 + x_2$	f_1	f_2	f'
Row 1	0	0	0	0	0	0
Row 2	x_1	0	x_1	0	$f_2\ (= x_1)$	$f_2\ (= x_1)$
Row 3	0	x_2	x_2	0	$f_2\ (= x_2)$	0
Row 4	x_1	x_1	0	0	0	0
Row 5	x_1	x_2	$x_1 + x_2$	$f_1\ (= x_1 + x_2)$	$f_1\ (= x_1 + x_2)$	0

The following pairs can be seen to be counter-examples in GF(2^2): $(x_1', x_2') = (01, 00)$ and $(x_1'', x_2'') = (00, 01)$. For these pairs we have: $f_1(x_1', x_2') = 00 = f_1(x_1'', x_2'')$, $f_2(x_1', x_2') = 01 = f_2(x_1'', x_2'')$, but $f'(x_1', x_2') = 01 \neq 00 = f'(x_1'', x_2'')$. This counter-example has been generated by looking at Rows 2 and 3: one of the technical challenges we solve is to systematically come up with counter-examples when arbitrary number of atoms and functions are involved.

Finally, consider the function f'':

$$f''(x_1, x_2) = \left\{ \begin{array}{ll} x_1 & \text{if } x_2 = 0 \\ x_2 & \text{if } x_1 = 0 \text{ and } x_2 \neq 0 \\ 0 & \text{otherwise} \end{array} \right\} = (1 + x_2^{q-1})x_1 + (1 + x_1^{q-1})x_2 \qquad (6)$$

Now the table looks like:

	x_1	x_2	$x_1 + x_2$	f_1	f_2	f''
Row 1	0	0	0	0	0	0
Row 2	x_1	0	x_1	0	x_1	x_1
Row 3	0	x_2	x_2	0	x_2	x_2
Row 4	x_1	x_1	0	0	0	0
Row 5	x_1	x_2	$x_1 + x_2$	$x_1 + x_2$	$x_1 + x_2$	0

Writing f'' as a combination of f_1, f_2:

	x_1	x_2	$x_1 + x_2$	f_1	f_2	f''
Row 1	0	0	0	0	0	0
Row 2	x_1	0	x_1	0	$f_2\ (= x_1)$	$f_2\ (= x_1)$
Row 3	0	x_2	x_2	0	$f_2\ (= x_2)$	$f_2\ (= x_2)$
Row 4	x_1	x_1	0	0	0	0
Row 5	x_1	x_2	$x_1 + x_2$	$f_1\ (= x_1 + x_2)$	$f_1\ (= x_1 + x_2)$	0

When we "collapse" the table to just the functions we have:

	f_1	f_2	$f_1 + f_2$	f''
Row 1	0	0	0	0
Row 2	0	f_2	f_2	f_2
Row 3	f_1	f_1	0	0

Now we claim that f'' is a function of f_1 and f_2 alone. In fact this can be verified easily:

$$f'' = f_1 + f_2 \tag{7}$$

In this particular case we observe that f'' is pseudo-linear in f_1, f_2. We actually prove the general result that if the target function is a function of the input functions, then it is a pseudo-linear function of the input functions.

10.4.1.2 A Basis for Pseudo-Linear Functions In this section we fix a field $\mathcal{F}_q$ of size $q = 2^m$.

Let D stand for all linear expressions (including zero) in n variables, say $x_1, x_2, ..., x_n$ (the unordered collection will be referred to as X). We define the set of **elementary pseudo-linear (**EPSELIN**)** polynomials to be all polynomials of the form

$$\prod_{l \in J}(1 + l(\vec{x})^{q-1}) \cdot \prod_{l \in \mathcal{D} \backslash J} l(\vec{x})^{q-1} \cdot p(\vec{x}) \tag{8}$$

where $p(\vec{x})$ is in D, and J is any subset of D such that it is closed under addition, i.e. J is a subspace of D. We also include the zero polynomial amongst the elementary pseudo-linear polynomials. Note that if $\mathcal{D} \backslash J$ included a linearly-dependent term of J, then the above polynomial reduces to zero in $\mathcal{F}_q$.

Generalizing (and specializing) the earlier definition of a guard, we will refer to expressions of the form

$$\prod_{l \in J}(1 + l(\vec{x})^{q-1}) \cdot \prod_{l \in \mathcal{D} \backslash J} l(\vec{x})^{q-1} \tag{9}$$

as **guards**.

For the next definition, we will require that the n variables be ordered by their indices. Thus x_1 is considered to be of lesser index then x_2, and so on. This also induces a lexicographic ordering on all equal-sized subsets of the n variables X.

An elementary pseudo-linear polynomial with the above notation will be called a **reduced elementary pseudo-linear (**REPSELIN**)** polynomial if it satisfies the following:

1. Let r be the rank of J ($r \leq \min(n, |J|)$).

2. Let R be the lexicographically greatest set of r variables occuring in J which can be expressed in terms of smaller indexed variables (or just zero) when J is set to zero. This for example, can be accomplished by considering a row-echelon normal form of J.

3. None of the variables in R occur in $p(\vec{x})$.

To justify this definition, we note that if an elementary pseudo-linear polynomial is not reduced, then it is equivalent to a reduced one.

One implication of the above definition is that if $p(\vec{x})$ is non-zero then it itself cannot be in J. Recall, J is closed under addition, by definition of EPSELIN-polynomials. Let r be the rank of J. Let $\bar{J}$ be the r sized subset of J which forms a basis of J, and which define the variables R by the row-echelon normal form of J. Thus, all $l(\vec{x})$ in J *must have at least one variable from* R. Thus, $p(\vec{x})$ cannot be in J.

Finally, we define a REPSELIN-polynomial to be a **basic pseudo-linear** polynomial if the linear term $p(\vec{x})$ is just a variable from X. Note that since the basic polynomial is REPSELIN, from item (3) above it follows that this variable is not from R.

Next we argue that any pseudo-linear polynomial can be expressed as a (xor-) sum of basic pseudo-linear polynomials. For this, we just need to show that any pseudo-linear polynomial of the form

$$\prod_{l \in \mathcal{D} \setminus J} l(\vec{x})^{q-1} \cdot p(\vec{x}) \tag{10}$$

where J is a subset of D, can be expressed as sum of EPSELIN-polynomials. This follows easily by induction on the size of J, and by noting that pseudo-linear polynomials with guards

$$\prod_{l \in \mathcal{D} \setminus J} l(\vec{x})^{q-1} \cdot \prod_{l \in J} (1 + l(\vec{x})^{q-1}) \tag{11}$$

such that $\mathcal{D} \setminus J$ includes a linearly dependent expression of J are identically zero.

We will now show that the basic pseudo-linear polynomials actually form a *basis* for pseudo-linear polynomials. Before that we need some more notation.

Let $\mathcal{Q}(X)$ be the set of all basic pseudo-linear polynomials in variables X. Further, let $\mathcal{G}(X)$ be the set of all guards amongst these polynomials $\mathcal{Q}(X)$. Let $|\mathcal{G}(X)| = t$. The guards can then be named w.l.o.g. g_1, g_2, ...,g_t. Recall, for each guard g_i, there is associated a subset of variables X, namely R, that do not occur in any linear terms $p(\vec{x})$. We refer to all linear combinations of $X \setminus R$ as $\mathcal{P}_i(X)$, including the linear term zero. Let $|\mathcal{P}_i(X)| = s_i + 1$. (Note, $(s_i + 1)$ is two to the power size of the subset of variables associated with g_i.) The linear terms in $\mathcal{P}_i(X)$ can be named $p_i^j(\vec{x})$, j ranging from 0 to s_i (not to be confused with exponent). W.l.o.g., zero will always be $p_i^0(\vec{x})$.

Thus, any pseudo-linear function $\phi(\vec{x})$ can be represented as a sum (over $\mathcal{F}_2$) of polynomials from $\mathcal{Q}(X)$, i.e.,

$$\phi(\vec{x}) \; = \; \sum_{i \in T} g_i(\vec{x}) \cdot p_i^{j(\phi,i)}(\vec{x}) \tag{12}$$

where T is a subset of $[1..t]$, and each $p_i^{j(\phi,i)}(\vec{x}) \in \mathcal{P}_i(X)$. In fact, we do not even need to take a subset T of $[1..t]$; all zero terms just imply that $j(\phi, i) = 0$, by our notation above that $p_i^0(\vec{x})$ is always taken to be zero. Thus the above representation of $\phi(\vec{x})$ is totally defined by the map $j(\phi, \cdot)$.

While we state and prove the following theorem only for large fields, as only for such fields are the basic pseudo-linear polynomials a basis, a slightly more complicated characterization can be given for smaller fields.

Lemma 20 (Basis) *For Fields of size $q > 2^n$, the basic pseudo-linear polynomials in n variables form a basis for pseudo-linear polynomials in n variables.*

Proof:

We have already shown above that any pseudo-linear polynomial can be represented as a sum of basic pseudo-linear polynomials, in fact defined by the map $j(\phi, \cdot)$ above. So, here we focus on showing that any pseudo-linear function $\phi(\vec{x})$ has a unique such representation.

So, for the sake of contradiction, suppose the everywhere zero function $\mathbf{0}$ has a non-zero representation, and let that be represented by the map $j(\mathbf{0}, i)$. Now consider any $\bar{i}$ where $j(\mathbf{0}, \bar{i}) \neq 0$, and lets call $p_i^{j(\mathbf{0},i)}(\vec{x})$ by just $p(x)$ $(\neq 0)$. In other words, this representation of $\mathbf{0}$ has the term

$$g_{\bar{i}} \cdot p(\vec{x})$$

Let,

$$g_{\bar{i}} = \prod_{l \in \mathcal{D} \backslash J} l(\vec{x})^{q-1} \cdot \prod_{l \in J} (1 + l(\vec{x})^{q-1}) \tag{13}$$

for some subspace $J \subseteq \mathcal{D}$. Let the rank of J be r. Let R be the lexicographically greatest set of r variables occuring in J which can be expressed in terms of smaller indexed variables when J is set to zero. Recall, by definition, none of the variables in R occur in $p(\vec{x})$.

Claim: With the set of equations J set to zero, we can solve for all linear expressions in $\mathcal{D} \backslash J$ to be non-zero, and hence also set $p(\vec{x})$ to non-zero.

This would first of all imply that all guards other than $g_i(\vec{x})$ evaluate to zero: if the guard $g_a(\vec{x})$ is given by subset $J_a \subseteq \mathcal{D}$ $(J_a \neq J)$, then if $J \backslash J_a$ is non-empty, we get that $\mathcal{D} \backslash J_a$ has an $l(\vec{x})$ from J which makes $l(x)^{q-1}$ zero, and if $J_a \backslash J$ is non-empty, we get that J_a has an $l(\vec{x})$ from $\mathcal{D} \backslash J$ which makes $(1 + l(\vec{x})^{q-1})$ zero.

Further, the guard $g_{\bar{i}}(\vec{x})$ will be non-zero, and hence $g_{\bar{i}}(\vec{x}) p(\vec{x})$ would be non-zero, and consequently the given representation $j(\mathbf{0}, i)$ leads to a non-zero function, a contradiction, which would prove the lemma.

Now, to prove the above claim, recall that J is closed under addition, and $p(\vec{x})$ is in $\mathcal{D} \backslash J$. Let r be the rank of J. Consider a basis $\bar{J}$ of a complementary subspace of J. If our underlying field is of size at least 2^{n-r}, we can set J to zero, and each $l_i(\vec{x})$ of $\bar{J}$ $(i \in [1..n-r])$ to e_i, where the e_i $(i \in [1..n-r])$ are linearly independent over $\mathcal{F}_2$. Thus, all linear expressions in $\mathcal{D} \backslash J$ evaluate to non-zero values, as any l in $\mathcal{D} \backslash J$ is a non-trivial linear combination of $\bar{J}$ plus an l' from J. $\square$

Note on small fields. In smaller fields some of the basic pseudo-linear polynomials, which are non-trivial functions in large fields, turn out to be identically zero. Thus the basis is smaller, but more complicated to characterize.

Lemma 21 (Homomorphism) *For any pseudo-linear functions $\phi_1(\vec{x})$ and $\phi_2(\vec{x})$, and for all $i \in [1..t]$,*

$$p_i^{j(\phi_1+\phi_2,i)} = p_i^{j(\phi_1,i)} + p_i^{j(\phi_2,i)} \tag{14}$$

Proof: Follows from the fact that the basic pseudo-linear polynomials form a basis for pseudo-linear polynomials. □

10.4.1.3 Interpolation Property for Pseudo-Linear Functions Before we prove the main theorem, we need a few more definitions and related lemmas.

Let $f_1, f_2, ..., f_k$ be k pseudo-linear functions in n variables X, over a field $\mathcal{F}_q$ ($q = 2^m$). Collectively, we will refer to these polynomials as F.

For any pseudo-linear polynomial $f(\vec{x})$ in X, let its representation in terms of the basis be given by $j(f, \cdot)$. Since each of the polynomials from F, i.e. $f_1(\vec{x}), f_2(\vec{x}), ..., f_k(\vec{x})$ is pseudo-linear, it be represented by $j(f_s, \cdot)$ ($s \in [1..k]$). Further, each linear combination of F is represented similarly.

We say that two guards $g_a(\vec{x})$ and $g_b(\vec{x})$ are F-**equivalent** if for every linear combination ϕ of functions from F, it is the case that $j(\phi, a) = 0$ iff $j(\phi, b) = 0$. In this case, we write $a \cong_F b$, which is an equivalence relation.

Lemma 22 *If a and b are F-equivalent then if for some subset $S \subseteq [1..k]$, the linear combination $\sum_{s\in S} p_a^{j(f_s,a)}$ is identically zero , then so is $\sum_{s\in S} p_b^{j(f_s,b)}$.*

The lemma follows by Lemma (21). Thus, if k' is the rank of $p_a^{j(f_s,a)}$ ($s \in [1..k]$), then it is also the rank of $p_b^{j(f_s,b)}$. In fact, we can take the exact same k' indices from ($s \in [1..k]$), w.l.o.g. $[1..k']$, to represent the basis for the k linear expressions, for both a and b.

Let $\mathcal{D}(F)$ denote the set of all linear combinations of functions in F.

For any function $f(\vec{x})$, and any set F of pseudo-linear functions in X, we say that $f(\vec{x})$ has the F-**interpolatable** property if it satisfies the following two conditions:

(i) $\forall i \in [1..t] : \exists \phi_\star \in \mathcal{D}(F) : j(f, i) = j(\phi_\star, i)$, and

(ii) For every $a, b \in [1..t]$ such that a and b are F-equivalent, w.l.o.g. by Lemma (22), let the first k' functions out of (k functions) $p_a^{j(f_s,a)}$ (out of $p_b^{j(f_s,b)}$), represent their basis (resp. for b). Then, if the $\phi_\star$ in (i) is given by $\sum c_s^a p_a^{j(f_s,a)}$ and $\sum c_s^b p_b^{j(f_s,b)}$, respectively for a and b, then for all $s \in [1..k']$, $c_s^a = c_s^b$.

Lemma 23 *If f is a pseudo-linear function in X, and f satisfies the F-interpolatable property, for some set F of pseudo-linear polynomials in X, then f is a pseudo-linear function of F.*

Proof: Indeed, consider $\hat{T} = [1..t] \;/\; \cong_F$, where t is the number of guards, i.e. $|\mathcal{G}(X)|$. We pick the smallest elements from $[1..t]$ to represent each equivalence class in $\hat{T}$. Define a function $h(\vec{x})$ to be the following:

$$h(\vec{x}) = \sum_{u \in \hat{T}} \prod_{\phi \in \mathcal{D}(F): j(\phi,u) \neq 0} \phi(\vec{x})^{q-1} \;\cdot \prod_{\phi \in \mathcal{D}(F): j(\phi,u)=0} (1 + \phi(\vec{x})^{q-1}) \;\cdot\; \phi_u(\vec{x}) \tag{15}$$

where for each u, ϕ_u is some function $\phi_\star$ satisfying the F-interpolatable property (i) above.

Now by definition, $h(\vec{x})$ is pseudo-linear in F. We now show that $h{=}f$, i.e. for all $\vec{x} \in (\mathcal{F}_q)^n$, $h(\vec{x}) = f(\vec{x})$. Fix any $\vec{x}^*$ in $(\mathcal{F}_q)^n$. Let $J \subseteq \mathcal{D}$, such that all linear functions in J evaluate to zero at $\vec{x}^*$, and all linear functions in $\mathcal{D} \backslash J$ evaluate to non-zero quantities at $\vec{x}^*$. Clearly, J is closed under addition, and hence J corresponds to a guard g_i. In other words, $g_i(\vec{x}^*) = 1$, and for all other $i' \in [1..t]$: $g_{i'}(\vec{x}^*) = 0$. Thus, $f(\vec{x}^*) = p_i^{j(f,i)}(\vec{x}^*)$, and similarly, for all $\phi \in \mathcal{D}(F)$, $\phi(\vec{x}^*) = p_i^{j(\phi,i)}(\vec{x}^*)$. By definition of i (i.e. g_i corresponding to J above, and hence $p_i^{j(\phi,i)} \in \mathcal{D} \backslash J$), it follows that $\phi(\vec{x}^*)$ is zero iff $j(\phi, i) = 0$.

Now, in equation (15), we show that the only u for which the "guards" evaluate to be non-zero (i.e. one), is the one corresponding to the equivalence class of i in $\hat{T}$ (say, u_i). In fact, for i (and its F-equivalent u_i) the "guards" indeed evaluate to 1. For all other i', if the "guards" evaluate to one, then by definition of F-equivalence, those i' are F-equivalent to i.

Thus, $h(\vec{x}^*) = \phi_{u_i}(\vec{x}^*)$, and since ϕ_{u_i} is pseudo-linear in X,

$$\phi_{u_i}(\vec{x}^*) = p_i^{j(\phi_{u_i},i)}(\vec{x}^*) = \sum_s c_s^{u_i} p_i^{j(f_s,i)}(\vec{x}^*) = \sum_s c_s^i p_i^{j(f_s,i)}(\vec{x}^*) = p_i^{j(\phi_i,i)}(\vec{x}^*). \tag{16}$$

Thus, $h(\vec{x}^*) = p_i^{j(f,i)}(\vec{x}^*)$, which is same as $f(\vec{x}^*)$. □

10.4.1.4 The Completeness Theorem for Pseudo-Linear Functions While the main completeness theorem below is stated and proven for only large finite fields, it holds for all finite fields of characteristic two.

Theorem 7 *Let $f_1, f_2, ..., f_k$ be k pseudo-linear functions in n variables X, over a field F_q ($q = 2^m$), such that $q > 2^n$. Collectively, we will refer to these polynomials as F. Let f be another pseudo-linear function in X. Then, if f is a function of F, then f is a pseudo-linear function of F.*

Proof: We show that if f is not a pseudo-linear function of F, which by Lemma (23) means that it does not satisfy at least one of F-interpolatable properties (i) or (ii), then f is not a function of F.

Since $f(\vec{x})$ is a pseudo-linear polynomial in X, let its representation in terms of the basis be given by $j(f, \cdot)$. Since each of the polynomials from F, i.e. $f_1(\vec{x})$, $f_2(\vec{x})$,...., $f_k(\vec{x})$ is pseudo-linear, it can also be represented by $j(f_s, \cdot)$ ($s \in [1..k]$). Further, each linear combination of F is represented similarly.

So, first consider the case where f does not satisfy (i). In other words, for some $i \in [1..t]$, for no linear combination ϕ of F (including zero) is $j(f,i)$ equal to $j(\phi,i)$. Thus, by Lemma (21), $p_i^{j(f,i)}$ is linearly independent of all $p_i^{j(f_s,i)}$ ($s \in [1..k]$). Let $J \subseteq \mathcal{D}$ correspond to the guard g_i. Thus, $p_i^{j(f,i)}$ and all $p_i^{j(f_s,i)}$ (for $s \in [1..k]$) are linearly independent of J. Let r be the rank of J, and $k' \leq k$ be the rank of $p_i^{j(f_s,i)}$ collectively. Since $p_i^{j(f,i)}$ is linearly independent of all $p_i^{j(f_s,i)}$, we have that $r + k' + 1 \leq n$. Now, the subspace corresponding to J set to zero has dimension $n - r$, and hence has q^{n-r} points. However, we are interested in points where all expressions in $\mathcal{D} \backslash J$ evaluate to non-zero values, which would guarantee that $g_i = 1$, and all other guards are zero. Recall the subspace $\mathcal{P}_i(X)$ generated by all variables not in the set R corresponding to guard g_i. Now, $\mathcal{D} \backslash J$ is a union of cosets of $(n - r)$ dimensional space $\mathcal{P}_i(X)$ shifted by subspace J. Consider a basis $\mathcal{B}$ for $\mathcal{P}_i(X)$, comprising of $p_i^{j(f,i)}$, a k'-ranked basis of $p_i^{j(f_s,i)}$, and $n - r - 1 - k'$ other linearly independent expressions $\mathcal{B}'$.

Assume the field $\mathcal{F}_q$ is of size at least 2^{n+1}, and hence has $n + 1$ linearly independent (over $\mathcal{F}_2$) elements e_i. Thus, for every injective map setting $\mathcal{B}$ to these e_i, there is a distinct solution to J being zero, and all of $\mathcal{D} \backslash J$ evaluating to non-zero values. Thus, there are at least $\binom{n+1}{n-r}(n - r)!$ such points in $(\mathcal{F}_q)^n$.

So, we fix $p_i^{j(f_s,i)}$ to e_s ($s \in [1..k']$; assume w.l.o.g. that the first k' formed the basis), and similarly fix the $\mathcal{B}'$ expressions to $e_{k'+1}$ to e_{n-r-1}. This still leaves at least $(n+1-(n-r-1))$ choices for $p_i^{j(f,i)}$. Thus, we have the situation that there are two points in $(\mathcal{F}_q)^n$ where f evaluates to different values, whereas F has the same value, and hence f cannot be a function of F.

Now, consider the case where f does satisfy condition (i), but condition (ii) is violated. In other words, for each $i \in [1..t]$, $p_i^{j(f,i)}$ is same as some $p_i^{j(\phi_\star,i)}$, but there exist a and b in $[1..t]$ which are F-equivalent, but the $\phi_\star$'s linear representation coefficients c_s differ for a and b. Again, we will demonstrate two points where F evaluate to the same value, but f evaluates to different values.

Again, let's assume that the underlying field is large enough to have at least k' linearly independent (over $\mathcal{F}_2$) elements, say e_i.

Now we have two sets, J_a corresponding to guard g_a, and J_b, corresponding to guard g_b. However, there is an easy solution for setting J_a to zero, and setting $p_a^{j(f_s,a)}$ ($s \in [1..k']$) to e_s. Similarly, there is a solution for setting J_b to zero and setting $p_a^{j(f_s,a)}$ ($s \in [1..k']$) to e_s. Thus, in both cases all f_s ($s \in [1..k]$) evaluate to the same value, but f being a different linear combination in the two cases, evaluates to different values. □

10.4.2 Iterated Composition of Pseudo-Linear Functions

In this section, we consider pseudo-linear functions which can take arguments, modeling oracles which are pseudo-linear functions of secret values and arguments. Thus, for instance it may be required to find if there exists a simulator which given access to functionalities which are pseudo-linear functions of secret parameters X and arguments supplied by simu-

lator/adversary, can compute a given a pseudo-linear function.

This generalizes the problem from the previous sections, where the simulator could not pass any arguments to the given functions. For simplicity, we will deal here with functions which only take a single argument, and thus all the functions can be written as $f_i(\vec{x}, y)$, each pseudo-linear in $\vec{x}$ and y.

So, given a collection of k pseudo-linear functions $F(X, y)$, we now define an **iterated composition of** F. Let $\mathcal{F}_q$ be the underlying field as before. An iterated composition σ of F is a length t sequence of pairs (t an arbitrary number), the first component of the s-th ($s \in [1..t]$) pair of σ being a function ϕ_s from F, and the second component an arbitrary function γ_s of $s-1$ arguments (over $\mathcal{F}_q$).

Given an iterated composition σ of F, one can associate a function f^σ of X with it as follows by induction. For σ of length one, f^σ is just $\phi_1(\vec{x}, \gamma_1())$, recalling that $\phi_1 \in F$. For σ of length t,

$$f^\sigma(\vec{x}) = \phi_t(\vec{x}, \gamma_t(f^{\sigma_{|1}}(\vec{x}), f^{\sigma_{|2}}(\vec{x}), ..., f^{\sigma_{|t-1}}(\vec{x}))) \tag{17}$$

where $\sigma_{|j}$ is the prefix of σ of length j.

Since, functions in n variables over $\mathcal{F}_q$ are just polynomials in n variables, there is a finite bound on t, after which no iterated composition of F can produce a new function of the n variables. The collection of all functions that can be obtained by iterated composition of F will be referred to as **terms(F)**. If we restrict γ_s to be pseudo-linear functions of their $s-1$ arguments, we will refer to the iterated composition as **pseudo-linear iterated composition of** F, and the corresponding collection of functions associated with such sequences as pseudo-linear iterated terms or **pl-terms(F)**. Note that in this case γ_1 is just zero.

Note that an arbitrary program can only compute a function of the terms, whereas an arbitrary pseudo-linear program can only compute a pseudo-linear function of the pseudo-linear terms. We would *like to show* that if a function f of terms(F) is a pseudo-linear function of X, then it is a pseudo-linear function of pl-terms(F). However, as we demonstrate in Section 10.4.2.1, this is not true in general, and a slight extension is required to the pl-terms, so as to enable probabilistic functions.

10.4.2.1 Example of Iterated Pseudo-Linear functions Consider the input function $f_1(x_1, y)$ and the target function $f(x_1)$ defined as follows:

$$f_1(x_1, y) = \left\{ \begin{array}{ll} x_1 & \text{if } y = x_1 \\ 0 & \text{otherwise} \end{array} \right\} = (1 + (x_1 + y)^{q-1})x_1 f(x_1) = x_1 \tag{18}$$

It is easy to see that the *iterated compositions* of $f_1(x_1, y)$ is just the single function $\mathbf{0}$, which outputs 0 on any input. However, it is possible to compute $f(x_1)$ by calling $f_1(x_1, y)$ as the following algorithm demonstrates:

Algorithm Simulate_f$^{f_1}()$

```
repeat for all non-zero elements y in F_q
  t ← f_1(x_1, y)
  if (t ?= y)
    return t
  y ← next y
end repeat block
return 0
```

In this example, we observe that the complexity of the algorithm is $O(q)$.

Now consider the following input and target functions:

$$f_1'(x_1, y) = \left\{ \begin{array}{ll} 0 & \text{if } y = 0 \text{ or } y = x_1 \\ x_1 & \text{otherwise} \end{array} \right\} = y^{q-1}(x_1 + y)^{q-1} x_1 f'(x_1) = x_1 \tag{19}$$

Here also the *iterated compositions* of $f_1'(x_1, y)$ is just the single function $\mathbf{0}$. Also, it is possible to compute $f(x_1)$ (with high probability) by calling $f_1'(x_1, y)$ as the following algorithm demonstrates:

Algorithm Simulate_f'$^{f_1'}()$

```
choose y randomly from F_q
t ← f_1(x_1, y)
return t
```

In this example, we observe that the complexity of the algorithm is $O(1)$, but it works with probability $1 - O(1/q)$: the probability of y being different from 0 and x_1. For this particular example, it is also possible to come up with an efficient deterministic algorithm - but systematically coming up with efficient deterministic algorithms in all cases where it's possible, seems to be a hard problem. We do show how to systematically come up with randomized efficient algorithms in all the cases where it is possible to do so.

10.4.2.2 Theorem for Iterated Pseudo-Linear Functions An iterated composition will be called an **extended pseudo-linear iterated composition of** F if γ_s is either a pseudo-linear function or a constant function $c(\vec{x})$ evaluating to an element c in $\mathcal{F}_q$. For each such c, the corresponding collection of functions associated with such sequences will be called **extended-pl-terms**(F, c).

We will also need to refine the definition of terms(F), by restricting to terms obtained within some T iterated compositions, for some positive integer T. Thus, **terms**$_T(F)$ will

stand for the collection of functions obtained by iterated compositions of F of length less than T. In particular we will be interested in T which is bounded by polynomials in $\log q$ and/or n, the number of variables in X.

Theorem 8 (Main) *Let $f_1, f_2, ..., f_k$ be k pseudo-linear functions in n variables X and an additional variable y, over a field $\mathcal{F}_q$ such that $q > 2^{2n}$. Collectively, we will refer to these polynomials as $F(X, y)$. Let T be a positive integer less than $2^n (< \sqrt{q})$. Let f be another pseudo-linear function in X. Then, if f is a function of* $\mathbf{terms}_T(F(X, y))$*, then f can be defined as a pseudo-linear probabilistic function of* $\mathbf{extended\text{-}pl\text{-}terms}(F(X, y), Seed)$*, where the probability is over Seed chosen uniformly from $\mathcal{F}_q$, and for each $\vec{x}$ the probability of this definition of f being correct is at least $1 - 1/\sqrt{q}$.*

Similar to Section 10.4.1.3, we first state an interpolatable property which is a sufficient condition for a pseudo-linear function of X to be a pseudo-linear function of pl-terms(F).

Recall the functions in F now have an additional argument y. As before, $\mathcal{D}(G)$, for any set of functions G will denote the set of all linear combinations (over $\mathcal{F}_2$) of functions from G. Below we define the class $\mathcal{I}^i(F)$ of pseudo-linear functions in X, for i an arbitrary natural number. In fact, since the inductive definition will sometimes use functions in both X and y, we will just define this class as pseudo-linear functions in X and y, though for different y, they would evaluate to the same value. In other words, for an arbitrary guard $g_a(\vec{x})$, which corresponds to a subset $J \subseteq \mathcal{D}(X)$ (J is closed under addition), there are many **super-guards** when viewed as a function of X and y, namely with subsets $J' \subseteq \mathcal{D}(X, y)$ (J' closed under addition) such that $J \subseteq J'$ *and* $(\mathcal{D}(X) \backslash J) \subseteq (\mathcal{D}(X, y) \backslash J')$. Thus, for all these super-guards, a pseudo-linear function $\phi(X)$ will have the same $j(\phi, \cdot)$ value (see Section 10.4.1.2).

However, and more importantly, with y set to some linear expression $l(\vec{x}) \in \mathcal{P}_a(X)$ (including zero), *exactly one* of these (super-)guards has the property that $J'_{y|l(\vec{x})} = J$ (Note the subscript $y|l(\vec{x})$ means $l(\vec{x})$ is substituted for every occurrence of y in J'). This particular J' is given by

$$J' \;=\; \mathcal{D}(J, \{y + l(\vec{x})\}) \tag{20}$$

In this case we say that this super-guard of g_a is consistent with $y + l(\vec{x}) = 0$. The super-guard corresponding to $J' = J$ will be called the **degenerate super-guard** of $g_a(\vec{x})$.

Now we define the pseudo-linear function which is the composition of f_s and h, i.e. $f_s \circ h$, where f_s is a pseudo-linear function in X and y, and h is a pseudo-linear function in X, by defining its components in the basis for pseudo-linear functions. For any guard $g_i(\vec{x})$ (of functions in X), let $g_I(\vec{x}, y)$ be the unique (super-) guard, mentioned in the previous paragraph, which is consistent with y set to $p_i^{j(h,i)}$ (note the map j here is for guards corresponding to X, and in general it will be clear from context whether we are referring to map j for guards corresponding to X or X, y). Then, define

$$p_I^{j(f_s \circ h, I)}(\vec{x}, y) = p_I^{j(f_s, I)}(\vec{x}, p_i^{j(h,i)}(\vec{x}, y)) \tag{21}$$

Further, for all I' which are super-guards of i, we set $p_{I'}^{j(f_s \circ h, I')}$ to be the same value (as $f_s \circ h$ is only a function of X). Note that since each p is just a linear function, this implies that each component of $f_s \circ h$ is a linear function of X (and hence X, y). In particular, $(f_s \circ h)(\vec{x}) = f_s(\vec{x}, h(\vec{x}))$.

Define **Compose**$(F(X,y), H(X))$, where $F(X,y)$ are a set of pseudo-linear functions in X, y and $H(X)$ is a set of pseudo-linear functions in X, to be the set of all functions $f_s \circ h$, where $f_s \in F(X,y)$ and $h \in H(X)$.

For each pseudo-linear function f_s of X and y, we also need to define a pseudo-linear function (in X) called **degenerate(f_s)**, which for each guard $g_a(\vec{x})$, defines the corresponding p function using its degenerate super-guard. Thus,

$$p_a^{j(\text{degenerate}(f_s),a)}(\vec{x}) = p_I^{j(f_s,I)}(\vec{x}, 0), \tag{22}$$

where I is the degenerate super-guard of g_a.

Now, we are ready to define the iterated pseudo-linear functions. Define

$$\mathcal{I}^0(F) = \mathcal{D}(\text{Compose}(F, \text{degenerate}(F)))\mathcal{I}^{i+1}(F) = \mathcal{D}(\mathcal{I}^i(F) \cup \text{Compose}(F, \mathcal{I}^i(F))), \text{ for } i \geq 0. \tag{23}$$

Since, these functions are just polynomials over finite fields (in fact defined over F_2), the above iteration reaches a fix-point at an i bounded by a function only of n. We will denote the fix-point by just $\mathcal{I}(F)$.

Now, we generalize the definitions of F-equivalence and F-interpolatable from Section 10.4.1.3. Two guards $g_a(\vec{x})$ and $g_b(\vec{x})$ are said to be F^*-equivalent if for every $\phi(\vec{x})$ in $\mathcal{I}(F)$, it is the case that $j(\phi, a) = 0$ iff $j(\phi, b) = 0$.

The definition of F^*-interpolatable property is same as the F-interpolatable property except that $\mathcal{D}(F)$ is replaced by $\mathcal{I}(F)$.

Instead of the closure $\mathcal{I}(F)$, it will also be useful to define the following set of functions

$$\mathcal{C} = \bigcup_i \text{Compose}(F, \mathcal{I}^i(F)) \cup \text{Compose}(F, \text{degenerate}(F)), \tag{24}$$

and it is easy to see that $\mathcal{I}(F)$ is just the linear closure of $\mathcal{C}$.

Lemma 24 *If f is a pseudo-linear function of n variables X over a field $\mathcal{F}_q$, and f satisfies the F^*-interpolatable property, for some set F of pseudo-linear polynomials in X, y, then f can be defined as a pseudo-linear probabilistic function of* **extended-pl-terms**$(F(X,y), Seed)$*, where the probability is over Seed chosen uniformly from $\mathcal{F}_q$, and for each $\vec{x}$ the probability of this definition of f being correct is at least $1 - 2^n/q$.*

Proof: The proof is similar to the proof of Lemma 23, but there is a small difference due to the probabilistic nature of this lemma.

Consider $\hat{T} = [1..t] \;/\; \cong_F$, where t is the number of guards, i.e. $|\mathcal{G}(X)|$. We pick the smallest element from $[1..t]$ to represent each equivalence class in $\hat{T}$. Define a function $h(\vec{x})$

to be the following:

$$h(\vec{x}) = \sum_{u \in \hat{T}} \prod_{\phi \in \mathcal{D}(F): j(\phi,u) \neq 0} \phi(\vec{x})^{q-1} \cdot \prod_{\phi \in \mathcal{D}(F): j(\phi,u)=0} (1 + \phi(\vec{x})^{q-1}) \cdot \phi_u(\vec{x}) \tag{25}$$

where for each u, ϕ_u is some function $\phi_\star \in \mathcal{C}$ satisfying the F^*-interpolatable property (i).

Now by definition, $h(\vec{x})$ is pseudo-linear in $\mathcal{C}$. Now, the proof that $h{=}f$, i.e. for all $\vec{x} \in (\mathcal{F}_q)^n$, $h(\vec{x}) = f(\vec{x})$, is identical to the proof in Lemma 23. However, h is pseudo-linear only on $\mathcal{C}$, whereas we need to show a function pseudo-linear in **extended-pl-terms**$(F(X,y), \text{Seed})$.

Observe that for any f_s, exactly one of the following cases hold, for all y linearly independent of $\vec{x}$:

Case 1: $f_s(\vec{x}, y) = \text{degenerate}(f_s)(\vec{x})$

Case 2: $f_s(\vec{x}, y) = \text{degenerate}(f_s)(\vec{x}) + y$

Now define a function $\hat{h}(\vec{x}, c)$ to be same as h, except every occurrence of degenerate$(f_s)(\vec{x})$ is replaced by either of the following:

$$\begin{cases} f_s(\vec{x}, c) & \text{in Case 1} \\ f_s(\vec{x}, c) + c & \text{in Case 2} \end{cases} \tag{26}$$

(recall, f_s is a function of X and y, whereas degenerate(f_s) is a function of only X). Then, it is easy to see that $\hat{h}(\vec{x}, c)$ is in **extended-pl-terms**(F, c). We next show that for every $\vec{x}$, with c chosen uniformly from $\mathcal{F}_q$, probability that $\hat{h}(\vec{x}, c) = h(\vec{x})$ is at least $1 - 2^n/q$. For each $\vec{x}$, one and only one guard g_s is satisfied. the probability that for this guard, $c = l(\vec{x})$, for some $l(\vec{x}) \in \mathcal{P}_s(X)$ is at most $1/q$. Hence, by union bound, over all possible $l(\vec{x})$, the probability that c equals any $l(\vec{x})$ is at most $2^n/q$, as $|X| = n$. These are the only cases in which degenerate$(f_s)(\vec{x})$ may differ from $f_s(\vec{x}, c)$ or $f_s(\vec{x}, c) + c$, as the case may be. □

Proof:[of Theorem 8]

For the sake of leading to a contradiction, suppose that f is not a pseudo-linear probabilistic function of **extended-pl-terms**$(F(X,y), Seed)$. Then by Lemma 24, f does not satisfy the F^*-interpolatable property. Hence, as in Theorem 7, we have two cases. Before we go into the analysis of the two cases, we recall a few relevant definitions, and state some useful properties.

Recall from Section 10.4.1.2, that $\mathcal{Q}(X)$ is the set of all basic pseudo-linear polynomials in variables X, and, $\mathcal{G}(X)$ is the set of all guards amongst these polynomials $\mathcal{Q}(X)$. Further, $|\mathcal{G}(X)| = t$. Also, recall for each guard g_s its corresponding set R from its REPSELIN representation, and the corresponding subspace $\mathcal{P}_s(X)$.

Also, recall the super-guards $g_I(\vec{x}, y)$ corresponding to guards $g_i(\vec{x})$. Thus, if J corresponded to g_i, then some J' such that $J \subseteq J' \subseteq \mathcal{D}(X,y)$, corresponds to super-guard g_I. Further, $(\mathcal{D}(X) \backslash J) \subseteq (\mathcal{D}(X,y) \backslash J')$. Hence, if some $y + l(\vec{x})$ is in J', we can w.l.o.g take as the corresponding R' (of g_I) to be $R \cup \{y\}$. Thus, in such cases $p_I(\vec{x}, y)$ are just linear

expressions in $X \backslash R$. If on the other hand, for no $l(\vec{x})$ it is the case that $y + l(\vec{x})$ is in J', then $R' = R$, and $p_I(\vec{x}, y)$ will be a linear expression in $(X \backslash R) \cup \{y\}$.

Now we are ready to analyze the two cases.

Case 1: First, consider the case where f does not satisfy property (1) of F^*-interpolatable.

Then, it is the case that there exists an $s \in [1..t]$, such that for every linear polynomial ϕ in $\mathcal{I}(F)$, $j(f,s) \neq j(\phi,s)$. Thus, by Lemma 21, $p_s^{j(f,s)}$ is linearly independent of all $p_s^{j(\phi,s)}$, with $\phi \in \mathcal{C}$.

Let $J \subseteq \mathcal{D}$ correspond to the guard g_s. Thus, $p_s^{j(f,s)}$, as well as all $p_s^{j(\phi,s)}$ ($\phi \in \mathcal{C}$) are linearly independent of J (see the paragraph after definition of REPSELIN polynomials). Let r be the rank of J, and k' be the rank of $p_s^{j(\phi,s)}$ collectively. Thus, $r + k' + 1 \leq n$. Consider a basis $\mathcal{B}$ of $\mathcal{P}_s(X)$ consisting of $p_s^{j(f,s)}$, a basis $\mathcal{B}''$ of $p_s^{j(\phi,s)}$, and another linearly independent set $\mathcal{B}'$ of expressions in $X \backslash R$ (of rank $n - r - k' - 1$).

Our aim is to demonstrate two different settings of X to values in $\mathcal{F}_q$, such that $f(\vec{x})$ has different values, while all of the $\mathbf{terms}_T(F(X,y))$ have the same value at the two settings. Now, fix a particular length T iterated composition σ of F. We will now show that each of $\gamma_t(f^{\sigma|1}(\vec{x}), f^{\sigma|2}(\vec{x}), ..., f^{\sigma|t-1}(\vec{x}))$, $t \in [1..T]$, as well as $f^{\sigma|t}(\vec{x})$ is a function of only $\mathcal{B}''$, and is independent of $p_s^{j(f,s)}$, and also independent of $\mathcal{B}'$ defined above. Thus in choosing the two different settings for X, we can first set the basis $\mathcal{B}''$ to some value, which will fix the $\gamma(...)$ values, and then we can set the $\mathcal{B}'$ and $p_s^{j(f,s)}$ to two different values, while assuring that all consistency requirements are met.

For the base case, $\gamma_1()$ is clearly not a function of $p_s^{j(f,s)}$, or $\mathcal{B}'$. Now, for the induction step, consider $f^{\sigma|t-1}(\vec{x})$. which is given by $\phi_{t-1}(\vec{x}, \gamma_{t-1}(f^{\sigma|1}(\vec{x}), f^{\sigma|2}(\vec{x}), ..., f^{\sigma|t-1}(\vec{x})))$. where ϕ_{t-1} is in F. Now, by induction the $\gamma_{t-1}(...)$ expression is not a function of $p_s^{j(f,s)}$ or $\mathcal{B}'$.

Now, it is possible that $\gamma_{t-1}(...)$ is equal to some $p_s^{j(\phi^*,s)}(\vec{x})$ ($\phi^* \in \mathcal{C}$), in which case $\phi_{t-1}(\vec{x}, \gamma_{t-1}(...))$ would just equal $p_I^{j(\phi_{t-1},I)}(\vec{x}, y)$, for I corresponding to the unique super-guard $g_I(\vec{x}, y)$ which is consistent with $y + p_s^{j(\phi^*,s)}(\vec{x}) = 0$. But, $p_I^{j(\phi_{t-1},I)}$ is either $p_s^{j(\phi^{**},s)}(\vec{x})$ ($\phi^{**} \in \mathcal{C}$) or such an expression plus y, by definition of $\mathcal{C}$ and definition of $p_I^{j(f_s \circ h,I)}(\vec{x}, y)$. In either case, it is a function of only $p_s^{j(\phi,s)}(\vec{x})$ ($\phi \in \mathcal{C}$) by induction.

If $\gamma_{t-1}(...)$ is not equal to any $p_s^{j(\phi^*,s)}(\vec{x})$ ($\phi^* \in \mathcal{C}$), we will show that we can choose $\vec{x}$ so as to assure that $\gamma_{t-1}(...)$ is not equal to any linear expression in $\mathcal{B}'$ (and $p_s^{j(f,s)}$) either, as $t < T < 2^n < \sqrt{q}$. In this case $\phi_{t-1}(\vec{x}, \gamma_{t-1}(...))$ returns $p_I^{j(\phi_{t-1},I)}(\vec{x}, y)$, where I corresponds to the degenerate super-guard of g_s given by $J' = J$. However, such $p_I^{j(\phi_{t-1},I)}(\vec{x}, y)$ is again either $p_s^{j(\phi^{**},s)}(\vec{x})$ ($\phi^{**} \in \mathcal{C}$) or such an expression plus y, since $\mathcal{C}$ includes Compose(F, degenerate(F)).

Now, we demonstrate the two different settings of X to values in $\mathcal{F}_q$. We first choose k' linearly independent (over $\mathcal{F}_2$) values in $\mathcal{F}_q$ and set the basis $\mathcal{B}''$ to these values, so that all expressions $p_s^{j(\phi,s)}$ ($\phi \in \mathcal{C}$) are non-zero. As explained above, the values $\gamma_t(...)$ are then fixed, and let this set of values along with zero be collectively called Γ. Next, we inductively assign values to the basis $\mathcal{B}'$(of size $n - r - k' - 1$) as follows. Let this basis be given by $l_1(\vec{x})$, ..., $l_{n-r-k'-1}(\vec{x})$. For $l_1(\vec{x})$, we pick any value in $\mathcal{F}_q$ which is not equal to any value in $\mathcal{D}(\mathcal{B}'')$

$+ \Gamma$, where the sum of two sets is defined naturally. For, the induction step, we choose for $l_i(\vec{x})$ a value in $\mathcal{F}_q$ which is not equal to any value in $\mathcal{D}(\mathcal{B}'' \cup \{l_1(\vec{x}), ..., l_{i-1}(\vec{x})\}) + \Gamma$.

Since $p_s^j(f,s)(\vec{x})$ is linearly independent of $\mathcal{B}''$ (as well as $\mathcal{B}'$), we choose a value for it which is not equal to any value in $\mathcal{D}(\mathcal{B}'' \cup \{l_1(\vec{x}), ..., l_{n-r-k'-1}(\vec{x})\}) + \Gamma$. Further we have at least two choices for it, given that $\mathcal{F}_q \geq 2 + 2^{n-1-r} \times (T+1)$. This also proves our claim that Γ is never equal to any linear expression in $\mathcal{B}' \cup \{p_s^{j(f,s)}\}$. Further no linear combination of $\mathcal{B}'$ and $\mathcal{B}''$ will be zero. Then, we can choose the R variables corresponding to the guard g_s, which by definition of R are given in terms of the variables already chosen, so that the guard g_s is true in both cases.

Case 2: Now consider the case where condition (i) holds, but condition (ii) of the F^*-interpolatable property fails to hold for f. In other words, for each $i \in [1..t]$, $p_i^{j(f,i)}$ is same as some $p_i^{j(\phi_*,j)}$ ((for $\phi_* \in \mathcal{C}$), but there exist a and b in $[1..t]$ which are F^*-equivalent, but the ϕ_*'s linear representation coefficients c_s differ for a and b. Again, we will demonstrate two points where $\text{terms}_T(F(X,y))$ evaluate to the same value, but f evaluates to different values.

We have two sets, J_a corresponding to guard g_a, and J_b, corresponding to guard g_b. Let $k' \leq n$ be the rank of $p_a^{j(\phi,a)}$ ($\phi \in \mathcal{C}$). Let r_a be the rank of J_a, and r_b be the rank of J_b. thus, $r_a + k' \leq n$, and $r_b + k' \leq n$. Let the R sets corresponding to guards g_a and g_b be called R_a and R_b respectively. Let $\mathcal{B}'_a$ be a basis for $X \backslash R_a$ excluding $\mathcal{D}(\mathcal{B}'')$, and similarly $\mathcal{B}'_b$ be a basis for $X \backslash R_b$ excluding $\mathcal{D}(\mathcal{B}'')$. We set the basis $\mathcal{B}''$ of $p_a^{j(\phi,a)}$ to linearly independent over GF2 values e_1 to $e_{k'}$. We set the basis of $p_b^{j(\phi,b)}$ also to the same values, recalling that the two bases, one for a and the other for b, can be chosen to have the same indices. Thus, all functions in $\mathcal{C}$ will have the same value when guards g_a or g_b are true. As in case 1, it follows that we can assure that by choosing $\mathcal{B}'_a$ and $\mathcal{B}'_b$ appropriately, each of $\gamma_t(f^{\sigma_{|1}}(\vec{x}), f^{\sigma_{|2}}(\vec{x}), ..., f^{\sigma_{|t-1}}(\vec{x}))$, $t \in [1..T]$, as well as $f^{\sigma_{|t}}(\vec{x})$ is only a function of $\mathcal{B}''$, and hence have the same values when guards g_a or g_b are true. However, since f has different linear combinations of $\mathcal{B}''$ at these two guards, it evaluates to different values. Further values for variables in R_a and R_b can be chosen so that guards g_a and g_b are indeed true. □

10.4.2.3 Allowing a Few Constants Let E be a set of linearly independent (over $\mathcal{F}_2$) elements of a field $\mathcal{F}_q$. Now, we *redefine* pseudo-linear polynomials where each linear term is as before defined over $\mathcal{F}_2$, but can in addition have an addend from E. The same rule also applies to all the linear terms in the guards. Then, we can prove the following theorem.

Theorem 9 *Let $f_1, f_2, ..., f_k$ be k pseudo-linear functions in n variables X, over a field $\mathcal{F}_q$ ($q = 2^m$), such that $q > 2^{n+|E|}$. Collectively, we will refer to these polynomials as F. Let f be another pseudo-linear function in X. Then, if f is a function of F, then f is a pseudo-linear function of F.*

Proof is similar to that of Theorem 7, in that we treat E as formal independent variables, and then in the proof of Theorem 7, we set these formal variables to E where we set the k' basis elements of $p_i^{j(f_s,i)}$ to e_s.

A similar version holds for the iterated composition Theorem 8.

10.4.3 Randomized Pseudo-Linear Functions

In this section we consider randomized pseudo-linear functions, or distributions over pseudo-linear families of pseudo-linear functions. A pseudo-linear family of pseudo-linear functions is given by a pseudo-linear function f' in variables $\vec{x}$ and $\vec{r}$, where the variables $\vec{r}$ parametrize the family. Given such a f', a randomized pseudo-linear function f (in $\vec{x}$) is given by choosing $\vec{r}$ uniformly and randomly.

The simulation question then becomes whether one can generate the target function distribution by sampling the input function distributions.

When we regard the $\vec{r}$ as formal variables, we can apply Lemma 20 to deduce that f' is expressible in terms of the basic pseudo-linear polynomials in $(\vec{x}, \vec{r})$. In particular,

$$f'(\vec{x}, \vec{r}) = \sum_{i \in T_{n+m}} g_i(\vec{x}, \vec{r}) \cdot p_i^{j(f',i)}(\vec{x}, \vec{r}) \tag{27}$$

Consider a guard g_i in just the space of the input variables $\vec{x}$, with associated set J, i.e. $g_i = \prod_{l \in \mathcal{D} \setminus J} l(\vec{x})^{q-1} \cdot \prod_{l \in J}(1 + l(\vec{x})^{q-1})$. Consider the set of super-guards $\mathbb{I}_i$ which extend J to $\mathcal{D} \cup \mathcal{D}(\vec{r})$ and each super-guard $I \in \mathbb{I}_i$ corresponds to a different subspace $J_r \subseteq \mathcal{D}(\vec{r})$ added to the subspace J (and then taking closure). Thus, we get a set of guards g_I $(I \in \mathbb{I}_i)$ corresponding to each guard g_i.

From now on, when clear from context, we will refer to the randomized function as $f(\vec{x}, \vec{r})$, to signify the random variables over which the distribution is defined.

We now show that given any randomized pseudo-linear function $f(\vec{x}, \vec{r})$, there is a randomized pseudo-linear function in just one random variable $\hat{r}$, such that it is statisitically indistinguishable from f. The *new* randomized pseudo-linear function $\hat{f}$ in just one random variable $\hat{r}$ and the same input variable set $\vec{x}$, is defined in the following way:

- The function $\hat{f}$ will only have non-zero p for guards involving only $\vec{x}$, i.e. p for guards involving $\hat{r}$ will be zero.
- For each guard g_i (with associated J), consider its extension super-guard $I_0 \in \mathbb{I}_i$ corresponding to $J_r = \{0\}$, In this case, J_{I_0} is just J. Suppose $p_{I_0}^{j(f,I_0)}(\vec{x}, \vec{r}) = l_1(\vec{x}) + l_2(\vec{r})$. If l_2 is not identically 0, then set $p_i^{j(\hat{f},i)}(\vec{x}) = \hat{r}$, otherwise set $p_i^{j(\hat{f},i)}(\vec{x}) = l_1(\vec{x})$.

Lemma 25 *Let $\log q > 2(\rho + m)$, where ρ is the number of random variables and m is the number of input variables in f. The distribution $f(\vec{x})$ is statistically indistinguishable from $\hat{f}(\vec{x})$ with advantage $< 1/\sqrt{q}$.*

Proof: Since there are at most $2^{\rho+m}$ different linear expressions in $\vec{x}$ and $\vec{r}$, by union bound, with probability at least $1 - 2^{\rho+m}/q > 1 - 1/\sqrt{q}$, all linear expressions $l_1(\vec{x}) + l_2(\vec{r})$ are

non-zero, when l_2 is not identically 0. In the rest of this proof, we therefore restrict ourselves to only on this overwhelming case.

Now consider any guard g_i in just $\vec{x}$. For any super-guard index $I \in \mathbb{I}_i - I_0$, at least one component $(1 + l(\vec{x}, \vec{r})^{q-1})$ in the super-guard will evaluate to 0, where $l(\vec{x}, \vec{r}) \in J$. Hence the guards corresponding to $(\mathbb{I}_i - I_0)$ all evaluate to 0.

For the index I_0, no component of J_{I_0} is identically zero since it is composed of $\vec{x}$ only. The components $(l_1(\vec{x}) + l_2(\vec{r}))^{q-1}$, where l_2 is not identically zero, evaluate to 1, since these are non-zero by the earlier restriction. Hence they can be dropped off, and we are just left with the guard g_i over just the $\vec{x}$'s. Now, suppose $p_{I_0}^{j(f,I_0)}(\vec{x}, \vec{r}) = l_1(\vec{x}) + l_2(\vec{r})$. If l_2 is identically zero, then $p_{I_0}^{j(f',I_0)}(\vec{x}, \vec{r})$ is just a function of $\vec{x}$. If l_2 is not identically 0, then $p_{I_0}^{j(f',I_0)}(\vec{x}, \vec{r})$ is just a uniformly distributed random number. We can set $p_i^{j(\hat{f},i)}(\vec{x}) = \hat{r}$. These random numbers $\hat{r}$ do not have to be different for every guard g_i, since for any fixed $\vec{x}$, exactly one of the guards is going to be equal to 1 and the rest of them will be 0. □

We will refer to the functions of the form of $\hat{f}$ as *Simplified Randomized Pseudo-Linear* functions (SRPL). These are functions which can be expressed with guards from $\vec{x}$ only and just one random variable. Lemma 25 indicates that we can just focus on SRPL functions since any randomized pseudo-linear function is statistically close to an SRPL function.

Lemma 26 (Homomorphism) *For any SRPL functions $\phi_1(\vec{x})$ and $\phi_2(\vec{x})$, and for all $i \in [1..t]$,*

$$p_i^{j(\phi_1+\phi_2,i)} = p_i^{j(\phi_1,i)} + p_i^{j(\phi_2,i)} \tag{28}$$

with the rule that $\hat{r} + \cdot$ is re-written as $\hat{r}$.

Proof: Follows from the fact that for a fixed $\vec{x}$, exactly one of the guards evaluates to 1 and the rest evaluate to 0. Also, adding a uniformly distributed random number to any quantity yields a uniformly distributed random number.

□

10.4.3.1 Interpolation Property for SRPL Functions Let $f_1, f_2, ..., f_k$ be k SRPL functions in n variables X, over a field $\mathcal{F}_q$ ($q = 2^m$). Collectively, we will refer to these polynomials as F.

For any SRPL function $f(\vec{x})$ in X, let its representation in terms of the basis be given by $j(f, \cdot)$. Note that there is also one special index j for the random variable r as well. Let this index be denoted by $j(f, \cdot) = \$$ uniformly.

Since each of the polynomials from F, i.e. $f_1(\vec{x}), f_2(\vec{x}), ..., f_k(\vec{x})$ is SRPL, let it be represented by $j(f_s, \cdot)$ ($s \in [1..k]$). Further, each linear combination of F is represented similarly, with the rule that $r + \cdot$ is replaced by r in the sum (see lemma 26).

We say that two guards $g_a(\vec{x})$ and $g_b(\vec{x})$ are F-**equivalent** if for every linear combination ϕ of functions from F, it is the case that $j(\phi, a) = 0$ iff $j(\phi, b) = 0$ *and* $j(\phi, a) = \$$ iff $j(\phi, b) = \$$. In this case, we write $a \cong_F b$, which is an equivalence relation.

Lemma 27 *If a and b are F-equivalent then if for some subset $S \subseteq [1..k]$, the linear combination $\sum_{s\in S} p_a^{j(f_s,a)}$ is identically zero, then so is $\sum_{s\in S} p_b^{j(f_s,b)}$; also if for some subset $S \subseteq [1..k]$, the linear combination $\sum_{s\in S} p_a^{j(f_s,a)}$ is random, then so is $\sum_{s\in S} p_b^{j(f_s,b)}$.*

The Lemma follows by Lemma (26). Thus, if k' is the rank of $p_a^{j(f_s,a)}$ ($s \in [1..k]$), then it is also the rank of $p_b^{j(f_s,b)}$. In fact, we can take the exact same k' indices from ($s \in [1..k]$), w.l.o.g. $[1..k']$, to represent the basis for the k linear expressions, for both a and b.

Let $\mathcal{D}(F)$ denote the set of all linear combinations of functions in F.

For any function $f(\vec{x})$, and any set F of pseudo-linear functions in X, we say that $f(\vec{x})$ has the F**-interpolatable** property if it satisfies the following two conditions:

(i) $\forall i \in [1..t] : j(f,i) = \$ \vee\ (\exists \phi_\star \in \mathcal{D}(F)\ :\ j(f,i) = j(\phi_\star, i))$, and

(ii) For every $a, b \in [1..t]$ such that a and b are F-equivalent,

either $j(f,a) = j(f,b) = \$$, in which case we set $j(\phi_\star, i) = \$$;

or, the following holds: w.l.o.g. by Lemma (27), let the first k' functions out of (k functions) $p_a^{j(f_s,a)}$ (out of $p_b^{j(f_s,b)}$), represent their basis (resp. for b). Then, if the $\phi_\star$ in (i) is given by $\sum c_s^a p_a^{j(f_s,a)}$ and $\sum c_s^b p_b^{j(f_s,b)}$, respectively for a and b, then for all $s \in [1..k']$, $c_s^a = c_s^b$.

Lemma 28 *Let $X \in (\mathcal{F}_q)^n$. If f is an SRPL function in X, and f satisfies the F-interpolatable property, for some set F of SRPL functions in X, then there exists a probabilistic poly-time (in $\lg q$) algorithm S^F, such that the distribution $f(X)$ is statistically indistinguishable from the distribution $S^F(X)$.*

Proof: Indeed, consider $\hat{T} = [1..t] \ / \cong_F$, where t is the number of guards, i.e. $|\mathcal{G}(X)|$. We pick the smallest elements from $[1..t]$ to represent each equivalence class in $\hat{T}$. Define a random variable $h(\vec{x})$ to be the following:

$$h(\vec{x}) \stackrel{\text{R}}{\leftarrow} S^F(\vec{x}) \tag{29}$$

where for each u, ϕ_u is some function $\phi_\star$ satisfying the F-interpolatable property above and algorithm S^F as follows:

Algorithm $S^F(\vec{x})$

```
for all u ∈ [1...t]/ ≅_F
  guardfound := true
  for all φ ∈ D(F)
    γ_0 ←R φ(x⃗); γ_1 ←R φ(x⃗)
    if γ_0 ≠ γ_1 then israndom := true else israndom := false
    if ( j(φ,u) = $ and israndom := true
       or j(φ,u) = 0 and γ_0 = 0 and israndom := false
       or j(φ,u) ≠ 0 and γ_0 ≠ 0 and israndom := false )
      then continue for loop
      else guardfound := false; exit for loop
  if guardfound = true then result ←R φ_u(x⃗); exit for loop
    else continue for loop
return result
```

We now show that $h \approx f$, i.e. for all $\vec{x} \in (\mathcal{F}_q)^n$, $h(\vec{x}) \approx f(\vec{x})$. Fix any $\vec{x}^*$ in $(\mathcal{F}_q)^n$. Let $J \subseteq \mathcal{D}$, such that all linear functions in J evaluate to zero at $\vec{x}^*$, and all linear functions in $\mathcal{D}\backslash J$ evaluate to non-zero quantities at $\vec{x}^*$. Clearly, J is closed under addition, and hence J corresponds to a guard g_i. In other words, $g_i(\vec{x}^*) = 1$, and for all other $i' \in [1..t]$: $g_{i'}(\vec{x}^*) = 0$. Thus, $f(\vec{x}^*) = p_i^{j(f,i)}(\vec{x}^*)$, and similarly, for all $\phi \in \mathcal{D}(F)$, $\phi(\vec{x}^*) = p_i^{j(\phi,i)}(\vec{x}^*)$. By definition of i (i.e. g_i corresponding to J above, and hence $p_i^{j(\phi,i)} \in \mathcal{D}\backslash J$), it follows that $\phi(\vec{x}^*)$ is zero iff $j(\phi, i) = 0$, and $\phi(\vec{x}^*)$ is random iff $j(\phi, i) = \$$.

Now, in the algorithm S^F, we show that the only u for which the "guards" evaluate to be non-zero (i.e. one), is the one corresponding to the equivalence class of i in $\hat{T}$ (say, u_i). In fact, for i (and its F-equivalent u_i) the "guards" indeed evaluate to 1: the reasoning is same as in the proof of Lemma 23 for the non-random values; in the random case, when we call $\phi(\vec{x})$ twice, the responses should be different with probability $1 - 1/q$, hence $\gamma_0 \neq \gamma_1$ would evaluate to $true$ - in any non-random case, $\gamma_0 \neq \gamma_1$ would evaluate to $false$ with probability one. For all other i', if the "guards" evaluate to one, then by definition of F-equivalence, those i' are F-equivalent to i.

Thus, $h(\vec{x}^*) \approx \phi_{u_i}(\vec{x}^*)$, and since ϕ_{u_i} is pseudo-linear in X,

$$\phi_{u_i}(\vec{x}^*) = p_i^{j(\phi_{u_i},i)}(\vec{x}^*) \approx \sum_s c_s^{u_i} p_i^{j(f_s,i)}(\vec{x}^*) = \sum_s c_s^i p_i^{j(f_s,i)}(\vec{x}^*) = p_i^{j(f,i)}(\vec{x}^*). \tag{30}$$

Thus, $h(\vec{x}^*) \approx p_i^{j(f,i)}(\vec{x}^*)$, which is same as $f(\vec{x}^*)$. □

10.4.3.2 The Completeness Theorem for SRPL Functions

Theorem 10 *Let $f_1, f_2, ..., f_k$ be k SRPL functions in n variables X, over a field $\mathcal{F}_q$ ($q = 2^m$), such that $q > 2^n$. Collectively, we will refer to these polynomials as F. Let f be another SRPL function in X. Then if there exists a probabilistic poly-time (in $\lg q$) algorithm S^F, such that the distribution $f(X)$ is statistically indistinguishable from the distribution $S^F(X)$, then f is F-interpolatable.*

Proof: We show that if f does not satisfy at least one of F-interpolatable properties (i) or (ii), then f is not efficiently simulatable using F. Then from this theorem and Lemma 28, it will follow that deciding whether an efficient simulator exists is equivalent to checking F-interpolatability, which can be done in computational time independent of $\lg q$.

Since $f(\vec{x})$ is a pseudo-linear polynomial in X, let its representation in terms of the basis be given by $j(f, \cdot)$. Since each of the polynomials from F, i.e. $f_1(\vec{x})$, $f_2(\vec{x})$,...., $f_k(\vec{x})$ is pseudo-linear, it can also be represented by $j(f_s, \cdot)$ ($s \in [1..k]$). Further, each linear combination of F is represented similarly.

So, first consider the case where f does not satisfy (i). In other words, for some $i \in [1..t]$, $j(f, i)$ is neither \$, nor for any linear combination ϕ of F (including zero) is $j(f, i)$ equal to $j(\phi, i)$. Thus, by Lemma (21), $p_i^{j(f,i)}$ is linearly independent of all the non-random $p_i^{j(f_s,i)}$ (w.l.o.g let's say $s \in [1..\hat{k}]$). Let $J \subseteq \mathcal{D}$ correspond to the guard g_i. Thus, $p_i^{j(f,i)}$ and all $p_i^{j(f_s,i)}$ (for $s \in [1..\hat{k}]$) are linearly independent of J. Let r be the rank of J, and $k' \leq \hat{k}$ be the rank of $p_i^{j(f_s,i)}$ collectively. Since $p_i^{j(f,i)}$ is linearly independent of all $p_i^{j(f_s,i)}$, we have that $r + k' + 1 \leq n$. Now, the subspace corresponding to J set to zero has dimension $n - r$, and hence has q^{n-r} points. However, we are interested in points where all expressions in $\mathcal{D} \backslash J$ evaluate to non-zero values, which would guarantee that $g_i = 1$, and all other guards are zero. Recall the subspace $\mathcal{P}_i(X)$ generated by all variables not in the set R corresponding to guard g_i. Now, $\mathcal{D} \backslash J$ is a union of cosets of $(n - r)$ dimensional space $\mathcal{P}_i(X)$ shifted by subspace J. Consider a basis $\mathcal{B}$ for $\mathcal{P}_i(X)$, comprising of $p_i^{j(f,i)}$, a k'-ranked basis of $p_i^{j(f_s,i)}$, and $n - r - 1 - k'$ other linearly independent expressions $\mathcal{B}'$.

Assume the field $\mathcal{F}_q$ is of size at least 2^{n+1}, and hence has $n + 1$ linearly independent (over $\mathcal{F}_2$) elements e_i. Thus, for every injective map setting $\mathcal{B}$ to these e_i, there is a distinct solution to J being zero, and all of $\mathcal{D} \backslash J$ evaluating to non-zero values. Thus, there are at least $\binom{n+1}{n-r}(n - r)!$ such points in $(\mathcal{F}_q)^n$.

So, we fix $p_i^{j(f_s,i)}$ to e_s ($s \in [1..k']$; assume w.l.o.g. that the first k' formed the basis), and similarly fix the $\mathcal{B}'$ expressions to $e_{k'+1}$ to e_{n-r-1}. This still leaves at least $(n+1-(n-r-1))$ choices for $p_i^{j(f,i)}$. Thus, we have the situation that there are two points in $(\mathcal{F}_q)^n$ where f evaluates to different values, whereas F has the same value. When we extend this argument to distributions, we observe that the $\hat{k}$ functions which are non-random have a constant distribution (their value is fixed given just $\vec{x}$) and also f is a constant distribution. The functions in F other than these $\hat{k}$ functions are uniform distribution. Therefore, collectively the counter-example generated above are also *easily detectably different* distributions, and hence f cannot be a function of F.

Now, consider the case where f does satisfy condition (i), but condition (ii) is violated. In other words, for each $i \in [1..t]$, $p_i^{j(f,i)}$ is same as some $p_i^{j(\phi_\star,i)}$, but there exist a and b in

$[1..t]$ which are F-equivalent, but one of the following two cases arise:

Case 1: $j(f,a) \neq \$$ and $j(f,b) \neq \$$, but $\phi_\star$'s linear representation coefficients c_s differ for a and b.

Again, we will demonstrate two points where F evaluate to the same value, but f evaluates to different values.

Again, let's assume that the underlying field is large enough to have at least k' linearly independent (over $\mathcal{F}_2$) elements, say e_i.

Now we have two sets, J_a corresponding to guard g_a, and J_b, corresponding to guard g_b. However, there is an easy solution for setting J_a to zero, and setting $p_a^{j(f_s,a)}$ $(s \in [1..k'])$ to e_s. Similarly, there is a solution for setting J_b to zero and setting $p_a^{j(f_s,a)}$ $(s \in [1..k'])$ to e_s. Thus, in both cases all f_s $(s \in [1..k])$ evaluate to the same value, but f being a different linear combination in the two cases, evaluates to different values.

Case 2: $j(f,a) = \$ \neq j(f,b)$.

In this case, we construct a counter-example in exactly the same manner as in Case 1. For guard g_a, we observe that f will follow a uniformly random distribution, whereas with guard g_b, f has a constant distribution. On the other hand the inputs have statistically indistinguishable distributions with the given counter-examples. Hence, f cannot be a function of F.

□

Completeness Theorem for Randomized Simulators and Iterated Composition of SRPL Functions In this section, we consider SRPL functions which can take arguments, modeling oracles which are SRPL functions of secret values and arguments. Thus, for instance it may be required to find if there exists a *randomized simulator* which given access to functionalities which are SRPL functions of secret parameters X and arguments supplied by simulator/adversary, can compute a given SRPL function.

This generalizes the problem from the previous sections, where the simulator could not pass any arguments to the given functions. For simplicity, we will deal here with functions which only take a single argument, and thus all the functions can be written as $f_i(\vec{x}, y)$, each SRPL in $\vec{x}$ and y.

So, given a collection of k SRPL functions $F(X, y)$, we now define an **iterated composition of** F. Let $\mathcal{F}_q$ be the underlying field as before. An iterated composition σ of F is a length t sequence of pairs (t an arbitrary number), the first component of the s-th ($s \in [1..t]$) pair of σ being a function ϕ_s from F, and the second component an arbitrary *randomized* function γ_s of $s-1$ arguments (over $\mathcal{F}_q$).

Given an iterated composition σ of F, one can associate a function f^σ of X with it as follows by induction. For σ of length one, f^σ is just $\phi_1(\vec{x}, \gamma_1())$, recalling that $\phi_1 \in F$. For σ of length t,

$$f^\sigma(\vec{x}) = \phi_t(\vec{x}, \gamma_t(f^{\sigma_{|1}}(\vec{x}), f^{\sigma_{|2}}(\vec{x}), ..., f^{\sigma_{|t-1}}(\vec{x}))) \tag{31}$$

where $\sigma_{|j}$ is the prefix of σ of length j.

Since, SRPL functions in n variables over $\mathcal{F}_q$ are just polynomials in n variables, there is a

finite bound on t, after which no iterated composition of F can produce a new SRPL function of the n variables. The collection of all functions that can be obtained by iterated composition of F will be referred to as **terms(F)**. If we restrict γ_s to be SRPL functions of their $s-1$ arguments, we will refer to the iterated composition as **SRPL iterated composition of** F, and the corresponding collection of functions associated with such sequences as SRPL iterated terms or **srpl-terms(F)**. Note that in this case γ_1 is just zero.

Note that an arbitrary randomized program can only compute a randomized function of the terms, whereas an arbitrary SRPL program can only compute an SRPL function of the SRPL terms. An iterated composition will be called an **extended SRPL iterated composition of** F if γ_s is either an SRPL function or a constant function $c(\vec{x})$ evaluating to an element c in $\mathcal{F}_q$. For each such c, the corresponding collection of functions associated with such sequences will be called **extended-srpl-terms(F, c)**.

We will also need to refine the definition of terms(F), by restricting to terms obtained within some T iterated compositions, for some positive integer T. Thus, $\mathbf{terms}_T(F)$ will stand for the collection of functions obtained by iterated compositions of F of length less than T. In particular we will be interested in T which is bounded by polynomials in $\log q$ and/or n, the number of variables in X.

Similar to Section 10.4.3.1, we first state an interpolatable property which is a sufficient condition for an SRPL function of X to be an SRPL function of srpl-terms(F).

Recall the functions in F now have an additional argument y. As before, $\mathcal{D}(G)$, for any set of functions G will denote the set of all linear combinations (over $\mathcal{F}_2$) of functions from G. Below we define the class $\mathcal{I}^i(F)$ of SRPL functions in X, for i an arbitrary natural number. In fact, since the inductive definition will sometimes use functions in both X and y, we will just define this class as SRPL functions in X and y, though for different y, they would evaluate to the same value. In other words, for an arbitrary guard $g_a(\vec{x})$, which corresponds to a subset $J \subseteq \mathcal{D}(X)$ (J is closed under addition), there are many **super-guards** when viewed as a function of X and y, namely with subsets $J' \subseteq \mathcal{D}(X, y)$ (J' closed under addition) such that $J \subseteq J'$ *and* $(\mathcal{D}(X) \backslash J) \subseteq (\mathcal{D}(X, y) \backslash J')$. Thus, for all these super-guards, a SRPL function $\phi(X)$ will have the same $j(\phi, \cdot)$ value (see Section 10.4.1.2).

However, and more importantly, with y set to some linear expression $l(\vec{x}) \in \mathcal{P}_a(X)$ (including zero), *exactly one* of these (super-)guards has the property that $J'_{y|l(\vec{x})} = J$ (Note the subscript $y|l(\vec{x})$ means $l(\vec{x})$ is substituted for every occurrence of y in J'). This particular J' is given by

$$J' \;=\; \mathcal{D}(J, \{y + l(\vec{x})\}) \tag{32}$$

In this case we say that this super-guard of g_s is consistent with $y + l(\vec{x}) = 0$. The super-guard corresponding to $J' = J$ will be called the **degenerate super-guard** of $g_a(\vec{x})$.

Now we define the SRPL function which is the composition of f_s and h, i.e. $f_s \circ h$, where f_s is a SRPL function in X and y, and h is a SRPL function in X, by defining its components in the basis for SRPL functions. For any guard $g_i(\vec{x})$ (of functions in X), let $g_I(\vec{x}, y)$ be the unique (super-) guard, mentioned in the previous paragraph, which is consistent with y set to $p_i^{j(h,i)}$ (note the map j here is for guards corresponding to X, and in general it will be

clear from context whether we are referring to map j for guards corresponding to X or X, y). Then, define

$$p_I^{j(f_s \circ h, I)}(\vec{x}, y) = p_I^{j(f_s, I)}(\vec{x}, p_i^{j(h,i)}(\vec{x}, y)) \tag{33}$$

Further, for all I' which are super-guards of i, we set $p_{I'}^{j(f_s \circ h, I')}$ to be the same value (as $f_s \circ h$ is only a function of X). Note that since each p is just a linear function, this implies that each component of $f_s \circ h$ is a linear function of X (and hence X, y). In particular, $(f_s \circ h)(\vec{x}) = f_s(\vec{x}, h(\vec{x}))$.

Define **Compose**$(F(X, y), H(X))$, where $F(X, y)$ are a set of SRPL functions in X, y and $H(X)$ is a set of SRPL functions in X, to be the set of all functions $f_s \circ h$, where $f_s \in F(X, y)$ and $h \in H(X)$.

For each SRPL function f_s of X and y, we also need to define a SRPL function (in X called **degenerate(f_s)**, which for each guard $g_a(\vec{x})$, defines the corresponding p function using its degenerate super-guard. Thus,

$$p_a^{j(\text{degenerate}(f_s), a)}(\vec{x}) = p_I^{j(f_s, I)}(\vec{x}, 0),$$

where I is the degenerate super-guard of g_a.

Now, we are ready to define the iterated SRPL functions. Define

$$\mathcal{I}^0(F) = \mathcal{D}(\text{Compose}(F, \text{degenerate}(F))) \tag{34}$$

$$\mathcal{I}^{i+1}(F) = \mathcal{D}(\mathcal{I}^i(F) \cup \text{Compose}(F, \mathcal{I}^i(F))), \text{ for } i \geq 0. \tag{35}$$

Since, these functions are just polynomials over finite fields (in fact defined over F_2), the above iteration reaches a fix-point at an i bounded by a function only of n. We will denote the fix-point by just $\mathcal{I}(F)$.

Now, we generalize the definitions of F-equivalence and F-interpolatable from Section 10.4.1.3. Two guards $g_a(\vec{x})$ and $g_b(\vec{x})$ are said to be F^*-equivalent if for every $\phi(\vec{x})$ in $\mathcal{I}(F)$, it is the case that $j(\phi, a) = 0$ iff $j(\phi, b) = 0$ and $j(\phi, a) = \$$ iff $j(\phi, b) = \$$.

The definition of F^*-interpolatable property is same as the F-interpolatable property except that $\mathcal{D}(F)$ is replaced by $\mathcal{I}(F)$.

Instead of the closure $\mathcal{I}(F)$, it will also be useful to define the following set of functions

$$\mathcal{C} = \bigcup_i \text{Compose}(F, \mathcal{I}^i(F)) \cup \text{Compose}(F, \text{degenerate}(F)), \tag{36}$$

and it is easy to see that $\mathcal{I}(F)$ is just the linear closure of $\mathcal{C}$.

Lemma 29 *If f is an SRPL function of n variables X over a field $\mathcal{F}_q$, and f satisfies the F^*-interpolatable property, for some set F of SRPL polynomials in X, y, then there exists a probabilistic poly-time (in $\lg q$) algorithm S^F, such that the distribution $f(X)$ is statistically indistinguishable from the distribution $S^F(X)$, with error at most $2^n/q$.*

Proof: The proof is similar to the proof of Lemma 23, but there is a small difference due to the probabilistic nature of this lemma.

Consider $\hat{T} = [1..t] \;/\; \cong_F$, where t is the number of guards, i.e. $|\mathcal{G}(X)|$. We pick the smallest element from $[1..t]$ to represent each equivalence class in $\hat{T}$. Define a function $h(\vec{x})$ to be the following:

$$h(\vec{x}) \xleftarrow{R} S^F(\vec{x}) \tag{37}$$

where S^F is as defined below. For each u, ϕ_u is some function $\phi_\star \in \mathcal{I}(F)$ satisfying the F^*-interpolatable property (i).

Algorithm $S^F(\vec{x})$

```
for all u ∈ [1...t]/ ≅_F
  guardfound := true
  for all φ ∈ I(F)
    γ0 <-R φ(x); γ1 <-R φ(x)
    if γ0 ≠ γ1 then israndom := true else israndom := false
    if   j(φ,u) = $ and israndom := true
      or j(φ,u) = 0 and γ0 = 0 and israndom := false
      or j(φ,u) ≠ 0 and γ0 ≠ 0 and israndom := false
      then continue for loop
      else guardfound := false; exit for loop
  if guardfound = true then result <-R φ_u(x); exit for loop
    else continue for loop
return result
```

Now, the proof that $h{=}f$, i.e. for all $\vec{x} \in (\mathcal{F}_q)^n$, $h(\vec{x}) = f(\vec{x})$, is identical to the proof in Lemma 23.

For the efficiency argument, we now show a simulator with calls to **extended-pl-terms**$(F(X,y), \text{Seed})$. Observe that for any f_s, exactly one of the following cases hold, for all y linearly independent of $\vec{x}$:

Case 1: $f_s(\vec{x}, y) = \text{degenerate}(f_s)(\vec{x})$

Case 2: $f_s(\vec{x}, y) = \text{degenerate}(f_s)(\vec{x}) + y$

Now define a function $\hat{h}(\vec{x}, c)$ to be same as h, except every occurrence of $\text{degenerate}(f_s)(\vec{x})$ is replaced by either of the following:

$$\begin{cases} f_s(\vec{x}, c) & \text{in Case 1} \\ f_s(\vec{x}, c) + c & \text{in Case 2} \end{cases} \tag{38}$$

(recall, f_s is a function of X and y, whereas degenerate(f_s) is a function of only X). Then, it is easy to see that $\hat{h}(\vec{x}, c)$ is in **extended-pl-terms**(F, c). We next show that for every $\vec{x}$, with c chosen uniformly from $\mathcal{F}_q$, probability that $\hat{h}(\vec{x}, c) = h(\vec{x})$ is at least $1 - 2^n/q$. For each $\vec{x}$, one and only one guard g_s is satisfied. the probability that for this guard, $c = l(\vec{x})$, for some $l(\vec{x}) \in \mathcal{P}_s(X)$ is at most $1/q$. Hence, by union bound, over all possible $l(\vec{x})$, the probability that c equals any $l(\vec{x})$ is at most $2^n/q$, as $|X| = n$. These are the only cases in which degenerate(f_s)($\vec{x}$) may differ from $f_s(\vec{x}, c)$ or $f_s(\vec{x}, c) + c$, as the case may be. □

Theorem 11 (Main) *Let $f_1, f_2, ..., f_k$ be k SRPL functions in n variables X and an additional variable y, over a field $\mathcal{F}_q$ such that $q > 2^{4n}$. Collectively, we will refer to these polynomials as $F(X, y)$. Let T be a positive integer less than $2^n (< q^{1/4})$. Let f be another SRPL function in X. Then if there exists a probabilistic poly-time (in $\lg q$) algorithm $S^{\mathbf{terms}_T(F(X,y))}$, such that the distribution $f(X)$ is statistically indistinguishable from the distribution $S^{\mathbf{terms}_T(F(X,y))}(X)$, then f is F^*-interpolatable.*

Proof: We show that if f does not satisfy at least one of F^*-interpolatable properties (i) or (ii), then f is not efficiently simulatable using F. Then from this theorem and Lemma 29, it will follow that deciding whether an efficient simulator exists is equivalent to checking F^*-interpolatability, which can be done in computational time independent of $\lg q$.

As in Theorem 7, we have two cases. In both cases we will show the non-existence of a probabilistic poly-time simulator (i.e. with for each $\vec{x}$, the probability of the definition being correct must be more than, say a very liberal, $q^{1/4}$). Before we go into the analysis of the two cases, we recall a few relevant definitions, and state some useful properties.

Recall from Section 10.4.1.2, that $\mathcal{Q}(X)$ is the set of all basic SRPL polynomials in variables X, and, $\mathcal{G}(X)$ is the set of all guards amongst these polynomials $\mathcal{Q}(X)$. Further, $|\mathcal{G}(X)| = t$. Also, recall for each guard g_s its corresponding set R from its REPSELIN representation, and the corresponding subspace $\mathcal{P}_s(X)$.

Also, recall the super-guards $g_I(\vec{x}, y)$ corresponding to guards $g_i(\vec{x})$. Thus, if J corresponded to g_i, then some J' such that $J \subseteq J' \subseteq \mathcal{D}(X, y)$, corresponds to super-guard g_I. Further, $(\mathcal{D}(X) \backslash J) \subseteq (\mathcal{D}(X, y) \backslash J')$. Hence, if some $y + l(\vec{x})$ is in J', we can w.l.o.g take as the corresponding R' (of g_I) to be $R \cup \{y\}$. Thus, in such cases $p_I(\vec{x}, y)$ are just linear expressions in $X \backslash R$. If on the other hand, for no $l(\vec{x})$ it is the case that $y + l(\vec{x})$ is in J', then $R' = R$, and $p_I(\vec{x}, y)$ will be a linear expression in $(X \backslash R) \cup \{y\}$.

Now we are ready to analyze the two cases.

Case 1: First, consider the case where f does not satisfy property (1) of F^*-interpolatable.

Then, it is the case that there exists an $s \in [1..t]$, such that for every linear polynomial ϕ in $\mathcal{I}(F)$, $j(f, s) \neq j(\phi, s)$, and further it is the case that $j(f, s)$ is not \$. Thus, by Lemma 21, $p_s^{j(f,s)}$ is linearly independent of all non-random $p_s^{j(\phi,s)}$, with $\phi \in \mathcal{C}$.

Let $J \subseteq \mathcal{D}$ correspond to the guard g_s. Thus, $p_s^{j(f,s)}$, as well as all $p_s^{j(\phi,s)}$ ($\phi \in \mathcal{C}$) are linearly independent of J (see the paragraph after definition of REPSELIN polynomials). Let r be the rank of J, and k' be the rank of $p_s^{j(\phi,s)}$ collectively. Thus, $r + k' + 1 \leq n$. Consider a basis $\mathcal{B}$ of $\mathcal{P}_s(X)$ consisting of $p_s^{j(f,s)}$, a basis $\mathcal{B}''$ of $p_s^{j(\phi,s)}$, and another linearly independent set $\mathcal{B}'$ of expressions in $X \backslash R$ (of rank $n - r - k' - 1$).

Note that since $j(f,s)$ is not random, it is a deterministic function of $\vec{x}$. Also, each term in $\mathbf{terms}_T(F(X,y))$ is an SRPL function, and hence is either a deterministic function of $\vec{x}$, or a random variable r. Thus, our aim is to demonstrate two different settings of X to values in $\mathcal{F}_q$, such that $f(\vec{x})$ has different values, while all of $\mathbf{terms}_T(F(X,y))$ have the same value (or distribution r) at the two settings. Now, fix a particular length T iterated composition σ of F. We will now show that each of $\gamma_t(f^{\sigma|1}(\vec{x}), f^{\sigma|2}(\vec{x}), ..., f^{\sigma|t-1}(\vec{x}))$, $t \in [1..T]$, as well as $f^{\sigma|t}(\vec{x})$ is a function of only $\mathcal{B}''$, and is independent of $p_s^{j(f,s)}$, and also independent of $\mathcal{B}'$ defined above. Thus in choosing the two different settings for X, we can first set the basis $\mathcal{B}''$ to some value, which will fix the $\gamma(...)$ distribution, and then we can set the $\mathcal{B}'$ and $p_s^{j(f,s)}$ to two different values, while assuring that all consistency requirements are met.

For the base case, $\gamma_1()$ is clearly not a function of $p_s^{j(f,s)}$, or $\mathcal{B}'$. Now, for the induction step, consider $f^{\sigma|t-1}(\vec{x})$. which is given by $\phi_{t-1}(\vec{x}, \gamma_{t-1}(f^{\sigma|1}(\vec{x}), f^{\sigma|2}(\vec{x}), ..., f^{\sigma|t-1}(\vec{x})))$. where ϕ_{t-1} is in F. Now, by induction the $\gamma_{t-1}(...)$ expression is not a function of $p_s^{j(f,s)}$ or $\mathcal{B}'$.

Now, it is possible that $\gamma_{t-1}(...)$ is equal to some $p_s^{j(\phi^*,s)}(\vec{x})$ ($\phi^* \in \mathcal{C}$), in which case $\phi_{t-1}(\vec{x}, \gamma_{t-1}(...))$ would just equal $p_I^{j(\phi_{t-1},I)}(\vec{x}, y)$, for I corresponding to the unique super-guard $g_I(\vec{x}, y)$ which is consistent with $y + p_s^{j(\phi^*,s)}(\vec{x}) = 0$. But, $p_I^{j(\phi_{t-1},I)}$ is either $p_s^{j(\phi^{**},s)}(\vec{x})$ ($\phi^{**} \in \mathcal{C}$) or such an expression plus y, by definition of $\mathcal{C}$ and definition of $p_I^{j(f_s \circ h,I)}(\vec{x}, y)$. In either case, it is a function of only $p_s^{j(\phi,s)}(\vec{x})$ ($\phi \in \mathcal{C}$) by induction.

If $\gamma_{t-1}(...)$ is not equal to any $p_s^{j(\phi^*,s)}(\vec{x})$ ($\phi^* \in \mathcal{C}$), we will *claim* that we can choose $\vec{x}$ so as to assure that $\gamma_{t-1}(...)$ is not equal to any linear expression in $\mathcal{B}'$ (and $p_s^{j(f,s)}$) either, with probability $> 1/q^{1/4}$ (the probability is over simulator's randomness and the randomness returned in the terms). In this case $\phi_{t-1}(\vec{x}, \gamma_{t-1}(...))$ returns $p_I^{j(\phi_{t-1},I)}(\vec{x}, y)$, where I corresponds to the degenerate super-guard of g_s given by $J' = J$. However, such $p_I^{j(\phi_{t-1},I)}(\vec{x}, y)$ is again either $p_s^{j(\phi^{**},s)}(\vec{x})$ ($\phi^{**} \in \mathcal{C}$) or such an expression plus y, since $\mathcal{C}$ includes Compose(F, degenerate(F)).

Now, we demonstrate the two different settings of X to values in $\mathcal{F}_q$. We first choose k' linearly independent (over $\mathcal{F}_2$) values in $\mathcal{F}_q$ and set the basis $\mathcal{B}''$ to these values, so that all expressions $p_s^{j(\phi,s)}$ ($\phi \in \mathcal{C}$) are non-zero.

Then, assuming the above claim holds, over T steps, the probability of some γ_t being a linear expression in $\mathcal{B}'$ is at most $T/q^{1/2}$. So, with probability $1 - T/q^{1/2}$, each γ_t is not equal to some linear expression in $\mathcal{B}'$, and hence the terms returned (i.e. ϕ_t) will be independent of $\mathcal{B}'$. Now, we show how to set $\mathcal{B}'$ and $p_s^{j(f,s)}$, so that the above claim holds. Let Γ be the set of values c (in the field) such that in some step (t in T), the probability of γ_{t-1} being c is more than $1/q^{1/2}$. (More formally, the proof should be done by maintaining an induction hypothesis about the claim, and building the set Γ inductively.) Note that $|\Gamma| < q^{1/2} \times (T+1)$.

Next, we inductively assign values to the basis $\mathcal{B}'$(of size $n - r - k' - 1$) as follows, so that the above claim holds. Let this basis be denoted by $l_1(\vec{x})$, ..., $l_{n-r-k'-1}(\vec{x})$. For $l_1(\vec{x})$, we pick any value in $\mathcal{F}_q$ which is not equal to any value in $\mathcal{D}(\mathcal{B}'') + \Gamma$, where the sum of two sets is defined naturally. For, the induction step, we choose for $l_\iota(\vec{x})$ a value in $\mathcal{F}_q$ which is not equal to any value in $\mathcal{D}(\mathcal{B}'' \cup \{l_1(\vec{x}), ..., l_{\iota-1}(\vec{x})\}) + \Gamma$.

Since $p_s^{j(f,s)}(\vec{x})$ is linearly independent of $\mathcal{B}''$ (as well as $\mathcal{B}'$), we choose a value for it which is not equal to any value in $\mathcal{D}(\mathcal{B}'' \cup \{l_1(\vec{x}), ..., l_{n-r-k'-1}(\vec{x})\}) + \Gamma$. Further we have at least two choices for it, given that $q \geq 2 + 2^{n-1-r} \times |\Gamma|$. Thus, no linear expression in $\mathcal{B}'$ or $p_s^{j(f,s)}$ is ever in Γ. Thus, f takes different values on these two settings os $\vec{x}$, whereas with probability $1 - T/q^{1/2} \geq 1 - 1/q^{1/4}$, all of $\mathbf{terms}_T(F(X, y))$ take the same value.
Case 2: Now consider the case where condition (i) holds, but condition (ii) of the F^*-interpolatable property fails to hold for f. This case is handled as in Theorem 10 but adapted with the analysis of Case 1 here. □

10.5 Proof Automation in the Universally Composable model

The Universally Composable (UC) framework is a formal system for proving security of computational systems such as cryptographic protocols. The framework describes two probabilistic games: The *real world* that captures the protocol flows and the capabilities of an attacker, and the *ideal world* that captures what we think of as a secure system. The notion of security asserts that these two worlds are essentially equivalent. A cryptographic motivation with an example is given in Section 10.5.2.

Formally, a proof of security in the UC model boils down to the following: as input, we are given two sets of algorithms:

1. Ideal Functionality: Set of algorithms $F = \{F_1, F_2, \cdots\}$
2. Real Protocol: Set of algorithms $P = \{P_1, P_2, \cdots\}$.

We say that P realizes F if it is possible to construct an algorithm S, called a simulator, that invokes the functions in F, such that the following holds:
For any PPT algorithm A, there exists a PPT algorithm S, such that for any PPT algorithm Z, the execution of A with calls to P is indistinguishable from the execution of S with calls to F.

We describe a language $L^{\$,\oplus,\mathsf{if}}$ in Table 2 for which we are able to develop a decision procedure to decide realizability of a given ideal functionality by a given real protocol.

Definition 25 ($L^{\$,\oplus,\mathsf{if}}$) *An Ideal Functionality F and a real protocol P are described in the language $L^{\$,\oplus,\mathsf{if}}$ if*

- *F is a set of programs $\{f_1(\vec{x}, \vec{y}), f_2(\vec{x}, \vec{y}), \cdots\}$.*
- *P is a single program $\{f(\vec{x})\}$.*

such that $f_1(\vec{x}, \vec{y}), f_2(\vec{x}, \vec{y}), \cdots$ and $f(\vec{x})$ are all described as $L^{\$,\oplus,\mathsf{if}}$ programs, as defined in Table 2.

The **semantics** of this language is that $\vec{x}$ is a set of inputs passed by the environment at the outset of execution and $\vec{y}$ is a set of parameters that the simulator is allowed to pass

Table 2: Programs in the Language $L^{\$,\oplus,\mathsf{if}}$.

(expressions)	AE	::=	$x_1 \mid x_2 \mid \cdots$	variables
	XE	::=	$AE \mid AE \oplus XE$	bitwise xor expression
	BE	::=	$\texttt{true} \mid (XE == XE) \mid BE \wedge BE \mid \neg BE$	boolean expression
(assignments)	a	::=	$x \leftarrow \$$	assign new random number
			$x := XE$	assign xor expression
(program)	π	::=	$a;$	single action
			$\pi a;$	sequence of actions
			$\texttt{if } BE \texttt{ then } \pi \texttt{ else } \pi$	conditional

to the functionalities. All the parameters and random numbers are represented as $\lg q$-bit strings, corresponding to elements in $\mathcal{F}_q$. The programs in F can be called in any order and an arbitrary number of times, whereas P is called only once. While fairly constrained, we provide a cryptographic example and motivation for this language in Section 10.5.2.

Theorem 12 (Completeness of $L^{\$,\oplus,\mathsf{if}}$) *There is a decision procedure, which given an Ideal Functionality F and a Real Protocol P described in the language $L^{\$,\oplus,\mathsf{if}}$, decides if P realizes F in the Universally Composable model.*

We prove Theorem 12 by starting off with the following lemma.

Lemma 30 *All the variables in an $L^{\$,\oplus,\mathsf{if}}$ program $f(\vec{z})$ are randomized pseudo-linear in $\vec{z}$.*

Proof: The proof is by structural induction on the grammar of expressions in $L^{\$,\oplus,\mathsf{if}}$. Since there are no loops in the language, we can assume wlog that no variable is assigned twice. In the base cases, the guards are derived according to the following rules ($[P]$ denotes the field polynomial corresponding to expression P) :

$$[x] = x, \text{ for atom } x \tag{39}$$
$$[XE_1 \oplus XE_2] = [XE_1] + [XE_2] \tag{40}$$
$$\tag{41}$$
$$[\texttt{true}] = 1 \tag{42}$$
$$[XE_1 == XE_2] = [XE_1 + XE_2]^{q-1} \tag{43}$$
$$[BE_1 \wedge BE_2] = [BE_1][BE_2] \tag{44}$$
$$[\neg BE] = 1 + [BE] \tag{45}$$

The conditional actions have an effect which can be viewed as follows for every relevant variable: $\texttt{if } BE \texttt{ then } x := XE_1 \texttt{ else } x := XE_2$. Then we will have $[x] = [BE][XE_1] + [\neg BE][XE_2]$, since $([BE], [\neg BE]) \in \{(0,1),(1,0)\}$.

It is easy to see that expressions constructed as above are pseudo-linear in the atoms. For the inductive case, xor-ing two pseudo-linear expressions again is a pseudo-linear expression. The only non-trivial case is the construction of conditional expressions from pseudo-linear expressions. We have to prove the following: *Any pseudo-linear polynomial raised to the power $q-1$ is a sum of* guard *expressions.* Given this the induction is straightforward.

To prove this recall that PLs can be expressed as sum of EPSELIN terms $\prod_{l\in\mathcal{D}/J} l(\vec{x})^{q-1} \cdot \prod_{l\in J}(1+l(\vec{x})^{q-1}) \cdot p(\vec{x})$. Observe that the product of any two distinct EPSELIN guards $\prod_{l\in\mathcal{D}/J_1} l(\vec{x})^{q-1} \cdot \prod_{l\in J_1}(1+l(\vec{x})^{q-1})$ and $\prod_{l\in\mathcal{D}/J_2} l(\vec{x})^{q-1} \cdot \prod_{l\in J_2}(1+l(\vec{x})^{q-1})$ is 0.

Therefore, we can write down any pseudo-linear polynomial (in $\vec{x}$) as:

$$GE = (EPS_1 + EPS_2 + \cdots) = (G_1.L_1 + G_2.L_2 + \cdots),$$

where the EPS_i's are EPSELIN terms, the G_i's are guards and the L_i's are the corresponding linear expressions (after gathering all the linear terms with the same G_i together). Now, for any substitution of the atoms $\vec{x}$, at most one of the G_i's is equal to 1 and the rest of the G_i's are 0. This lets us write:

$$(G_1.L_1 + G_2.L_2 + \cdots)^{q-1} = \begin{cases} 0 & \text{if all the } G_i(\vec{x})\text{'s are 0} \\ L_1(\vec{x})^{q-1} & \text{if } G_1(\vec{x}) = 1 \\ L_2(\vec{x})^{q-1} & \text{if } G_2(\vec{x}) = 1 \\ \cdots & \end{cases} \tag{46}$$

Hence this is exactly equal to the polynomial $(G_1.L_1^{q-1} + G_2.L_2^{q-1} + \cdots)$ which is a sum of guard expressions in the atoms. □

We now proceed to the main proof.

Proof:[Theorem 12] By Lemma 30, all the functions in P and F compute randomized pseudo-linear functions in the inputs. By Lemma 25, with negligible error, we can assume that these are given as SRPL functions.

Now, by Theorem 11, if f is simulatable using $\mathbf{terms}_T(F(X,y))$, with $T < q^{1/4}$, then f is F^*-interpolatable. F^*-interpolatability can be decided by computing $\mathcal{I}(F)$, which can be computed in time independent of $\lg q$. Further, by Lemma 29, if f is F^*-interpolatable, then there exists a probabilistic polynomial time (in $\lg q$) simulator.

□

10.5.1 Extension with Persistent States and Uninterpreted Functions

Consider an extension of the language $L^{\$,\oplus,\mathsf{if}}$, where we add a fixed number of variables to the Ideal world that are persistent across subroutine calls. Let us call this language $L^{\$,\oplus,\mathsf{if},\mathsf{state}}$. We describe the key ideas for developing a decision procedure for $L^{\$,\oplus,\mathsf{if},\mathsf{state}}$.

We first construct stateful iterated compositions of $L^{\$,\oplus,\mathsf{if},\mathsf{state}}$ subroutines. We construct a tree where each node is such a composition and its subtrees denote further compositions extending its own computation. The key observation is that there is only a finite number

of such nodes which are distinct modulo renaming of uniformly random quantities. In other words, the nodes fall into a finite number of equivalence classes modulo permutation of uniformly random quantities and hence represent the *same randomized algorithm*. An important property of these equivalence classes is that two members of a class lead to subtrees which are equivalent as sets.

This leads to the conclusion that there is a finite set of stateful iterated compositions. A considerably harder theorem is to prove completeness: if there is a probabilistic polytime simulator, then there is a simulator which is s stateful iterated composition of the subroutines.

10.5.1.1 Key ideas for encryption and signatures. The UC formulation of the encryption and signature primitives makes them expressible in very abstract terms. We leverage the fact that standard notions of security of these primitives have been shown to be equivalent to the UC formulation. Specifically, we express protocols in the *hybrid model*, where the concrete operations for encryptions and signatures are replaced by their ideal counterpart. The proofs of security translate due to the *Composition Theorem* supported by the UC framework.

Signatures. The UC formulation of signatures is given in Figure 24. In addition to the operations in $L^{\$,\oplus,\mathsf{if}}$, we need functions (*viz.*, s and v) and storage (for the records). For a single session of a protocol, the honest protocol participants only do a bounded number of signatures. However, the adversary may make an unbounded number of calls to the verification function - but this does not create any requirement for more storage. Hence the language $L^{\$,\oplus,\mathsf{if},\mathsf{state}}$ suffices for the storage part.

As regards the function variables s, v, v', observe that they do not have to satisfy any equation. Hence they can be treated as *uninterpreted* function symbols. For example $s(m)$ can be represented as the tuple $\langle \text{“}s\text{”}, m \rangle$, where “$s$” is a constant string. To support these entities, we only need to define tuples, constants and equality of tuples.

Public-Key Encryption. The UC formulation of PKE is given in Figure 25. The discussion on signatures carries over to PKE. In addition, we need to distinguish the ciphertexts being output on separate invocations of the Encryption subroutine. This can be done by tagging the ciphertexts with a uniformly random quantity generated at each invocation: $[r \leftarrow \$;\ c := \langle e', \mathbf{0}, r \rangle ;]$.

However, in contrast to the signature functionality, the adversary can induce a requirement for unbounded storage by calling the Encryption subroutine multiple times. The language $L^{\$,\oplus,\mathsf{if},\mathsf{state}}$ is only able to support a bounded number of such calls - hence we only have a conditional security proof. While for individual protocols it can be rather simple to prove that it suffices to show realizability for an adversary which makes only a bounded (or even single) number of calls to the hybrid encryption functionality, it is an interesting open problem to prove a structural meta-theorem which establishes the sufficiency of a bounded number of calls for a general class of protocols.

Functionality F_{sig}:

- Key Generation: Upon receiving a value $(\texttt{KeyGen}, sid)$ from some party S, verify that $sid = (S, sid')$ for some sid'. If not, then ignore the request. Else, hand $(\texttt{KeyGen}, sid)$ to the adversary. Upon receiving $(\texttt{Algorithms}, sid, s, v)$ from the adversary, where s and v are descriptions of feasible algorithms, output $(\texttt{Verification Algorithm}, sid, v)$ to S.
- Signature Generation: Upon receiving a value $(\texttt{Sign}, sid, m)$ from S, let $\sigma = s(m)$, and verify that $v(m, s) = 1$. If so, then output $(\texttt{Signature}, sid, m, \sigma)$ to S and record the entry (m, σ). Else, output an error message to S and halt.
- Signature Verification: Upon receiving a value $(\texttt{Verify}, sid, m, \sigma, v')$ from some party V, do: If $v' = v, v(m, \sigma) = 1$, and no entry (m, σ') for any σ' is recorded, then output an error message to S and halt. Else, output $(\texttt{Verified}, sid, m, v'(m, \sigma))$ to V.

Figure 24: Ideal Functionality for Signatures

Functionality F_{pke}:
The message space is $GF(2^m)$. Let $\mathbf{0}$ be the zero element of $GF(2^m)$.

- Key Generation: Upon receiving a value $(\texttt{KeyGen}, sid)$ from some party D, verify that $sid = (D, sid')$ for some sid'. If not, then ignore the request. Else, hand $(\texttt{KeyGen}, sid)$ to the adversary. Upon receiving $(\texttt{Algorithms}, sid, e, d)$ from the adversary, where e and d are descriptions of feasible algorithms, output $(\texttt{Encryption Algorithm}, sid, e)$ to D.
- Encryption: Upon receiving a value $(\texttt{Encrypt}, sid, m, e')$ from some party V, do: If $e' \neq e$, or the decryptor D is corrupted, output $e'(m)$. Else, let $c = e'(\mathbf{0})$ and record (m, c). Output $(\texttt{Ciphertext}, sid, c)$ to E.
- Decryption: Upon receiving a value $(\texttt{Decrypt}, sid, c)$ from D, do: If there is a recorded entry (m, c) for some m then return $(\texttt{Plaintext}, sid, m)$ to D. Else, return $(\texttt{Plaintext}, d(c))$.

Figure 25: Ideal Functionality for Public-Key Encryption

10.5.2 Example of Automation

Password-based key exchange is an important security problem which has been studied extensively in cryptographic research [BM93], and which brings out the power of the UC framework particularly well. Canetti et al [CHK+05] proposed an Ideal Functionality for password-based key exchange which is formally described in Figure 26.

Functionality $\mathcal{F}_{\mathbf{pwKE}}$

The functionality $\mathcal{F}_{\text{pwKE}}$ is parameterized by a security parameter k. It interacts with an adversary S and a set of parties via the following queries:

Upon receiving a query (NewSession, $sid, P_i, P_j, pw, role$) **from party** P_i**:**
Send (NewSession, $sid, P_i, P_j, role$) to S. In addition, if this is the first NewSession query, or if this is the second NewSession query and there is a record (P_j, P_i, pw'), then record (P_i, P_j, pw) and mark this record fresh.

Upon receiving a query (TestPwd, sid, P_i, pw') **from the adversary** S**:**
If there is a record of the form (P_i, P_j, pw) which is fresh, then do: If $pw = pw'$, mark the record compromised and reply to S with "correct guess". If $pw \neq pw'$, mark the record interrupted and reply with "wrong guess".

Upon receiving a query (NewKey, sid, P_i, sk) **from** S**, where** $|sk| = k$**:**
If there is a record of the form (P_i, P_j, pw), and this is the first NewKey query for P_i, then:

- If this record is compromised, or either P_i or P_j is corrupted, then output (sid, sk) to player P_i.
- If this record is fresh, and there is a record (P_j, P_i, pw') with $pw' = pw$, and a key sk' was sent to P_j, and (P_j, P_i, pw) was fresh at the time, then output (sid, sk') to P_i.
- In any other case, pick a new random key sk' of length k and send (sid, sk') to P_i.

Either way, mark the record (P_i, P_j, pw) as completed.

Figure 26: The password-based key-exchange functionality $\mathcal{F}_{\text{pwKE}}$

Consider two parties P_i and P_j that wish to come up with a common cryptographically strong key based on the fact that they share the same password. The idea is to capture the fact that modulo the adversary outright guessing the password exactly during an active session between the parties, it has no control (or information) on the key being generated. It is allowed to interrupt sessions by tampering with the messages being exchanged, but doing so only results in the parties ending up with different uniformly randomly distributed keys. If, however, the session is not interrupted, the parties end up with the same key which is

distributed uniformly and randomly and is not controlled by the adversary.

In the following discussion, we will overlook the session ids for simplicity although it is straightforward to add them. We describe a protocol Π_{ICPwKE}, in Figure 27, which is a candidate to realize $\mathcal{F}_{\text{pwKE}}$. This is a protocol based on the Ideal Cipher model [BPR00]. In the ideal cipher model, the results of two decryptions are the same if the key is identical. Otherwise, the results are uniformly and independently random. The protocol is symmetric from the perspective of both the participants - so we describe the actions of just one party P_i. Both parties get a password from the environment $\mathcal{E}$. Party P_i generates a random number r_1, encrypts it and sends the ciphertext c_1 to the peer. When it receives a response c_2', it first checks whether its own message was reflected. If so, it outputs a random key to the environment. Otherwise, it decrypts the response using its password pw_1 and xors the plaintext with r_1. The resulting quantity is output as its key to the environment.

Party P_i	Adv	Party P_j
$\mathcal{E}$		$\mathcal{E}$
$\downarrow pw_1$		$\downarrow pw_2$
$r_1 \leftarrow \$$		$r_2 \leftarrow \$$
$c_1 \leftarrow enc_{pw_1}(r_1)$		$c_2 \leftarrow enc_{pw_2}(r_2)$
	$\xrightarrow{c_1}$ $\xleftarrow{c_2}$	
	$\xleftarrow{c_2'}$ $\xrightarrow{c_1'}$	
`if` $(c_2' == c_1)$ `then` $sk_1 \leftarrow \$$		`if` $(c_1' == c_2)$ `then` $sk_2 \leftarrow \$$
`else`		`else`
$d_1 \leftarrow dec_{pw_1}(c_2')$		$d_2 \leftarrow dec_{pw_2}(c_1')$
$sk_1 \leftarrow r_1 \oplus d_1$		$sk_2 \leftarrow r_2 \oplus d_2$
$\downarrow sk_1$		$\downarrow sk_2$
$\mathcal{E}$		$\mathcal{E}$

Figure 27: Protocol for Password-based Key Exchange using Ideal Cipher.

Consider the following *ideal functionality for the ideal cipher* primitive. The functionality takes two arguments: a key and a plaintext. It has a table where each entry is a triplet (key, plaintext, ciphertext). The table is initially empty. It supports two subroutines: encrypt(key, plaintext) and decrypt(key, ciphertext). The **encrypt** subroutine, given input $(key, plaintext)$, generates a random number r, stores $(key, plaintext, r)$ in the table and outputs r. The **decrypt** subroutine, given input $(key, ciphertext)$, looks up if there is an entry $(key, p, ciphertext)$ in the table. If so, it outputs p. Otherwise, it generates a random number r, stores $(key, r, ciphertext)$ in the table and outputs r.

Now consider the real-world scenario where the adversary intercepts the first message c_1 and changes it to c_1' before transmitting to P_j. The adversary's action may involve querying the ideal cipher in the hybrid model. More importantly, if the password is weak, the adversary maybe able to guess the password, and hence a proper simulation would require the simulator to extract this password guess from the call to the ideal cipher, and use that in the TestPwd subroutine of $\mathcal{F}_{\text{PWKE}}$.

We now describe how our decision procedure automatically figures out such a simulator, as the languages for the real protocol (in the ideal cipher hybrid model) and for the ideal

functionality $\mathcal{F}_{\text{PWKE}}$ are as covered by our theorems and their extensions. First, however we need a theorem which states that it suffices to consider an adversary which makes only a single call to the hybrid ideal cipher functionality. Such a theorem is rather easy to prove for our particular protocol, but it is likely that such meta-theorems can be proved as general structural theorems. We can then inline a single ideal cipher call in a serialization of the protocol (the variables input by the adversary can be named, say kk and rr). Observe that the following operations are sufficient to describe the ideal cipher functionality: equality testing, conditional branches, random number generation and table storage and lookups. When we consider a constant number of calls to the ideal cipher, the table operations reduce to assignment statements and equality testing. The ideal functionality $\mathcal{F}_{\text{PWKE}}$ is clearly supported by our language extended as in Section 10.5.1. Finally, notice that the variables c_1' and c_2' are also available to the simulator. Thus the decision problem is whether the serialization of the protocol (with a single inlined ideal cipher call) can be obtained as a randomized iterated composition of $\mathcal{F}_{\text{PWKE}}$, where the simulator also has access to variables kk, rr, c_1' and c_2'.

10.6 Undecidable cases

In this section we describe one language L_{tab} for which we are able to prove that no algorithmic procedure exists to decide equivalence. In particular, this arises when unbounded table lookup/storage operations are allowed (not even random access, but just storing and detecting whether some string is there in the table), even when there are no arithmetic operations, loops in subroutines or random number generation. Formally, we have the following theorem:

Theorem 13 (An Undecidable System) *Let L_{tab} be a language with input / outputs to the environment, send / receives to the adversary, conditional with equality checking of strings and table storage/lookup. We are given a real protocol $\mathcal{P}$ and a ideal functionality $\mathcal{F}$ with all subroutines described in L_{tab}. We show that it is impossible to algorithmically decide whether $\mathcal{P}$ realizes $\mathcal{F}$.*

We describe the language L_{tab} formally in Table 3. Note that there is no arithmetic, logical operation or nonce generation in the language. On the other hand, strings can be of arbitrary length.

10.6.1 Analysis

Theorem 14 *The language $Sim_{tab} = \{(\mathcal{P}, \mathcal{F}) \mid \mathcal{P}$ realizes $\mathcal{F}$, $\mathcal{P}$ and $\mathcal{F}$ are described in $L_{tab}\}$ is undecidable .*

Proof: We reduce the problem $\{M \mid M$ is a Turing Machine which accepts the empty string $\epsilon\}$ to Sim_{tab}. Specifically, given the state transition representation of M, we construct a $(\mathcal{P}, \mathcal{F})$ instance whose membership in Sim_{tab} is equivalent to deciding whether M reaches an accepting state on input the empty string.

Table 3: L_{tab}: Language definition for the undecidable system

(atomic terms)	u	::=	x	atomic term variable
			s	string
(terms)	t	::=	y	term variable
			u	atomic term
			$t.t$	pair of terms (bounded)
(table)	τ	::=	tab	table
(actions)	a	::=	`send` t	send a term t
			`receive` y	receive term into variable y
			`output` t	output t to the environment
			$y := t$	assign t to y
			$store(t, \tau)$	store t in τ
(booleans)	b	::=	$t = t$	equality check
			$lookup(t, \tau)$	true iff t is in τ
(program)	π	::=	$a;$	single action
			$\pi a;$	sequence of actions
			`if` (b) `then` $\{\pi\}$	conditional
(real protocol)	$\mathcal{P}$	::=	π	one program
(ideal functionality)	$\mathcal{F}$	::=	$\{\pi, \pi, \cdots, \pi\}$	set of programs

Construction of $\mathcal{P}$.

The real protocol $\mathcal{P}$ has just the following subroutine $\mathcal{P}_1$: [`receive` x; `output` "*success*";]

Construction of $\mathcal{F}$.

The functionality $\mathcal{F}$ models construction of "cells" corresponding to individual cells in the configuration of a Turing Machine computation. $\mathcal{F}$ consists of several small subroutines which perform operations like populating the table with the initial cells, copying non transiting cells to the next configuration, performing state transition at the TM header, detecting acceptance state and declaring success. We give formal descriptions of the subroutines in Section 10.6.1.1. The informal description is as follows:

- $\mathcal{F}_{init}$: This is the subroutine which is called in the beginning. It stores some elements in the table corresponding to the empty string ϵ.
- $\mathcal{F}_{\rho}$: For each transition rule ρ, there is a corresponding subroutine which has the effect of producing the next configuration according to this transition rule.
- $\mathcal{F}_{begin}$: Subroutine that constructs the beginning of the current configuration.
- $\mathcal{F}_{\omega}$: Subroutine that constructs the end of the current configuration.
- $\mathcal{F}_{persist}$: Subroutine that carries over bits from previous configuration to the current one, if unaffected by state transistion.
- $\mathcal{F}_{accept}$: Subroutine that transitions to an accepting state.
- $\mathcal{F}_{success}$: Subroutine that declares success upon completing an accepting configuration.

Simulation for an M that accepts ϵ.

The construction is such that there is a way to simulate transition from one configuration to the next. We show that if the Turing Machine accepts the empty string then there is a way for the simulator to simulate the actions of this Turing Machine faithfully, leading to successful calling of the subroutine $\mathcal{F}_{success}$ - we give an explicit procedure below:

▷ First $\mathcal{F}_{init}$ is called to store the empty string into the table.

▷ Let $CurrentConfiguration \leftarrow EmptyConfiguration$

▷ Repeat the following:

– Call $\mathcal{F}_{begin}$ to construct the first cell of $NextConfiguration$

- If current cell is in the neighborhood of the state cell, call $\mathcal{F}_\rho$ with the corresponding transition rule ρ. This subroutine also pushes the "end" cell, if the state cell goes past it. Record *AcceptanceStateReached* if new state is an accepting state.
- Otherwise call $\mathcal{F}_{persist}$ to copy bit from corresponding cell in $CurrentConfiguration$ to $NextConfiguration$.
- If *AcceptanceStateReached* is true and "end" cell is reached, let set $CallSuccess$ to true.
- Otherwise, let $CurrentConfiguration \leftarrow NextConfiguration$

▷ Until $CallSuccess$ is true.

▷ Call $\mathcal{F}_{success}$ and halt.

Impossibility of simulation for an M that does not accept ϵ.

We show that it is impossible to register a configuration in the table, which does not follow from an already registered configuration. Thus if there is an accepting configuration in the table, then there must be a sequence of configurations in the table, beginning with the empty configuration and ending with accepting configuration, such that each configuration leads to the next by a single transition.

Every storage cell has the following structure:

```
Current Conf Id.  Current Cell Id.  Cell Element.  Next Cell Id.  Next Conf Id
```

Before any cell is stored in the table, the previous cell also has to be provided by the simulator. It is checked whether the immediate predecessor is already there or not: the corresponding Id's have to match up correctly - think of this as a 2-dimensional linked list, in which each configuration is a linked list and where `Next Cell Id` points to the next cell in the current configuration linked list and `Next Conf Id` points to the next configuration linked list. Each time a new configuration is being built, a new `Next Conf Id` has to be provided by the simulator. The functionality checks that it is new by first checking for presence and then storing it in the table as a type "confid" data. Similarly for new cells that go past the current tape length, there is checking and storing of a type "cellid" data. This combination of checks ensures the integrity of the 2-D linked list - in particular, there are no rogue pointers to elements in number of different configurations. Ensuring that the simulator provides the previous cell also "bootstraps" the cell records consistently - one cannot enter a new element if the previous element is not already there. Importantly, this ensures the following Lemma:

Lemma 31 *For a given* `Current Conf Id` *linked list, it is only possible to call a unique $\mathcal{F}_\rho$.*

This is because the simulator *has to* "bootstrap" up to the correct state cell following the pointers and only a unique transition works for a given state and cell after the state cell. Consequently, the following assertion holds:

Lemma 32 *If there is an "end" cell registered for a given* `Current Conf Id`*, the linked list of cells with this confid is the successor of some configuration in the table.*

This implies our original claim that if there is an accepting configuration in the table, then there must be a sequence of configurations in the table, beginning with the empty configuration and ending with accepting configuration, such that each configuration leads to the next by a single transition. This is a contradiction since no such sequence exists for the given M. Hence no simulator exists.

□

10.6.1.1 Ideal Functionality for the Undecidable Language This section describes the ideal functionality for the language subset which is undecidable.

Init.

$\mathcal{F}_{init}$: [

`receive` *trigger*;

`store` "$conf0$"."$cell0$"."$begin$"."$cell1$"."$conf1$", τ;

`store` "$conf0$"."$cell1$"."$qstart$"."$cell2$"."$conf1$", τ;

`store` "$conf0$"."$cell2$"." "."$cell3$"."$conf1$", τ;

`store` "$conf0$"."$cell3$"."end"."$cell4$"."$conf1$", τ;

`store` "$cellid$"."$cell0$", τ;

`store` "$cellid$"."$cell1$", τ;

`store` "$cellid$"."$cell2$", τ;

`store` "$cellid$"."$cell3$", τ;

`store` "$confid$"."$conf0$", τ;

`store` "$confid$"."$conf1$", τ;

]

Transition Rule. For a transition rule $\rho : (q, 0) \rightarrow (r, 1, R)$, we have the subroutine:

$\mathcal{F}_\rho$: [

`receive` $cell_1, cell_2, cell_3, cell_4, cell_5$;

`receive` $cell'_1, cell'_2, cell'_3, cell'_4, cell'_5$;

`parse` $cell_1$ `as` $\sigma.\beta_1.\ b_1.\ \beta_2.\sigma'$;

`parse` $cell_2$ `as` $\sigma.\beta_2.\ b_2.\ \beta_3.\sigma'$;

`parse` $cell_3$ `as` $\sigma.\beta_3.$ "q". $\beta_4.\sigma'$;

`parse` $cell_4$ `as` $\sigma.\beta_4.$ "$zero$". $\beta_5.\sigma'$;

`parse` $cell_5$ `as` $\sigma.\beta_5.\ b_4.\ \beta_6.\sigma'$;

`parse` $cell'_1$ `as` $\sigma'.\beta_1.\ b_1.\ \beta_2.\sigma''$;

`parse` $cell'_2$ `as` $\sigma'.\beta_2.\ b_2.\ \beta_3.\sigma''$;

`parse` $cell'_3$ `as` $\sigma'.\beta_3.$ "one". $\beta_4.\sigma''$;

`parse` $cell'_4$ `as` $\sigma'.\beta_4.$ "r". $\beta_5.\sigma''$;

`parse` $cell'_5$ `as` $\sigma'.\beta_5.\ b_4.\ \beta_6.\sigma''$;

`if` (none of $cell_1, cell_2, cell_3, cell_4, cell_5$ is in τ)
  `then output` "$failure$";

`if` ($cell'_1$ is not in τ)
  `then output` "$failure$";

`store` $cell'_2, cell'_3, cell'_4, cell'_5$ `in` τ;

]

For a transition rule $\rho : (q, 0) \rightarrow (r, 1, L)$, we have the subroutine:

$\mathcal{F}_\rho$: [
  receive $cell_1, cell_2, cell_3, cell_4, cell_5$;
  receive $cell'_1, cell'_2, cell'_3, cell'_4, cell'_5$;

  parse $cell_1$ as $\sigma.\beta_1.\ b_1.\ \beta_2.\sigma'$;
  parse $cell_2$ as $\sigma.\beta_2.\ b_2.\ \beta_3.\sigma'$;
  parse $cell_3$ as $\sigma.\beta_3.$ "q". $\beta_4.\sigma'$;
  parse $cell_4$ as $\sigma.\beta_4.$ "$zero$". $\beta_5.\sigma'$;
  parse $cell_5$ as $\sigma.\beta_5.\ b_4.\ \beta_6.\sigma'$;

  parse $cell'_1$ as $\sigma'.\beta_1.\ b_1.\ \beta_2.\sigma''$;
  parse $cell'_2$ as $\sigma'.\beta_2.$ "r". $\beta_3.\sigma''$;
  parse $cell'_3$ as $\sigma'.\beta_3.\ b_2.\ \beta_4.\sigma''$;
  parse $cell'_4$ as $\sigma'.\beta_4.$ "one". $\beta_5.\sigma''$;
  parse $cell'_5$ as $\sigma'.\beta_5.\ b_4.\ \beta_6.\sigma''$;

  if (none of $cell_1, cell_2, cell_3, cell_4, cell_5$ is in τ)
    then output "$failure$";
  if ($cell'_1$ is not in τ)
    then output "$failure$";
  store $cell'_2, cell'_3, cell'_4, cell'_5$ in τ;
]

Boundary Regions.

$\mathcal{F}_{begin}$: [
  receive $cell_1, cell_2$;
  receive $cell'_1, cell'_2$;

  parse $cell_1$ as $\sigma.\beta_1.$ "$begin$". $\beta_2.\sigma'$;
  parse $cell_2$ as $\sigma.\beta_2.\ b.\ \beta_3.\sigma'$;

  parse $cell'_1$ as $\sigma'.\beta_1.$ "$begin$". $\beta_2.\sigma''$;
  parse $cell'_2$ as $\sigma'.\beta_2.\ b.\ \beta_3.\sigma''$;

  if (none of $cell_1, cell_2$ is in τ)
    then output "$failure$";
  if ("$confid$".σ'' is in τ)
    then output "$failure$";
  store $cell'_1, cell'_2$ in τ;
  store "$confid$".σ'' in τ;
]

For a transition rule of the form $\omega : (q, \_) \rightarrow (r, 0, R)$ at the end of the tape:

$\mathcal{F}_{\omega}$: [

receive $cell_1, cell_2, cell_3, cell_4, cell_5$;

receive $cell'_1, cell'_2, cell'_3, cell'_4, cell'_5, cell'_6$;

parse $cell_1$ as $\sigma.\beta_1.\ b_1.\ \beta_2.\sigma'$;

parse $cell_2$ as $\sigma.\beta_2.\ b_2.\ \beta_3.\sigma'$;

parse $cell_3$ as $\sigma.\beta_3.$ "q". $\beta_4.\sigma'$;

parse $cell_4$ as $\sigma.\beta_4.$ " ". $\beta_5.\sigma'$;

parse $cell_5$ as $\sigma.\beta_5.$ "end". $\beta_6.\sigma'$;

parse $cell'_1$ as $\sigma'.\beta_1.\ b_1.\ \beta_2.\sigma''$;

parse $cell'_2$ as $\sigma'.\beta_2.\ b_2.\ \beta_3.\sigma''$;

parse $cell'_3$ as $\sigma'.\beta_3.$ "$zero$". $\beta_4.\sigma''$;

parse $cell'_4$ as $\sigma'.\beta_4.$ "r". $\beta_5.\sigma''$;

parse $cell'_5$ as $\sigma'.\beta_5.$ " ". $\beta_6.\sigma''$;

parse $cell'_6$ as $\sigma'.\beta_6.$ "end". $\beta_7.\sigma''$;

if (none of $cell_1, cell_2, cell_3, cell_4, cell_5$ is in τ)
  then output "$failure$";

if ($cell'_1$ is not in τ)
  then output "$failure$";

if ("$cellid$".β_7 is in τ)
  then output "$failure$";

store $cell'_2, cell'_3, cell'_4, cell'_5, cell'_6$ in τ;

]

Persistence of other bits.

$\mathcal{F}_{persist}$: [

receive $cell_1, cell_2$;

receive $cell'_1, cell'_2$;

parse $cell_1$ as $\sigma.\beta_1.\ b_1.\ \beta_2.\sigma'$;

parse $cell_2$ as $\sigma.\beta_2.\ b_2.\ \beta_3.\sigma'$;

parse $cell'_1$ as $\sigma'.\beta_1.\ b_1.\ \beta_2.\sigma''$;

parse $cell'_2$ as $\sigma'.\beta_2.\ b_2.\ \beta_3.\sigma''$;

if (none of $cell_1, cell_2$ is in τ)
  then output "$failure$";

if ($cell'_1$ is not in τ)
  then output "$failure$";

store $cell'_1$ in τ;

]

Detecting Machine Acceptance. For a transition rule $\rho : (q, 0) \rightarrow (q_{accept}, 1, R)$, we have the subroutine:

$\mathcal{F}_\rho$: [
receive $cell_1, cell_2, cell_3, cell_4, cell_5$;
receive $cell'_1, cell'_2, cell'_3, cell'_4, cell'_5$;

parse $cell_1$ as $\sigma.\beta_1.\ b_1.\ \beta_2.\sigma'$;
parse $cell_2$ as $\sigma.\beta_2.\ b_2.\ \beta_3.\sigma'$;
parse $cell_3$ as $\sigma.\beta_3.$ "q". $\beta_4.\sigma'$;
parse $cell_4$ as $\sigma.\beta_4.$ "$zero$". $\beta_5.\sigma'$;
parse $cell_5$ as $\sigma.\beta_5.\ b_4.\ \beta_6.\sigma'$;

parse $cell'_1$ as $\sigma'.\beta_1.\ b_1.\ \beta_2.\sigma''$;
parse $cell'_2$ as $\sigma'.\beta_2.\ b_2.\ \beta_3.\sigma''$;
parse $cell'_3$ as $\sigma'.\beta_3.$ "one". $\beta_4.\sigma''$;
parse $cell'_4$ as $\sigma'.\beta_4.$ "$qaccept$". $\beta_5.\sigma''$;
parse $cell'_5$ as $\sigma'.\beta_5.\ b_4.\ \beta_6.\sigma''$;

if (none of $cell_1, cell_2, cell_3, cell_4, cell_5$ is in τ)
then output "$failure$";
if ($cell'_1$ is not in τ)
then output "$failure$";
store $cell'_2, cell'_3, cell'_4, cell'_5$ in τ;
store "$AcceptanceStateReached.''.\sigma'$ in τ;
]

Similarly for a transition rule $\rho : (q, 0) \rightarrow (r, 1, L)$.

Subroutine to check consistency at the end:

$\mathcal{F}_{success}$: [
receive $cell_1, cell_2$;
receive $cell'_1, cell'_2$;

parse $cell_1$ as $\sigma.\beta_1.$ " ". $\beta_2.\sigma'$;
parse $cell_2$ as $\sigma.\beta_2.$ "end". $\beta_3.\sigma'$;

parse $cell'_1$ as $\sigma'.\beta_1.$ " ". $\beta_2.\sigma''$;
parse $cell'_2$ as $\sigma'.\beta_2.$ "end". $\beta_3.\sigma''$;

if (none of $cell_1, cell_2$ is in τ)
then output "$failure$";
if ($cell'_1$ is not in τ)
then output "$failure$";

if ("$AcceptanceStateReached$".σ' is in τ)
then output "$success$";
]

10.7 Summary and Outlook

Security primitives and protocols are deceptively concise. It is not too hard for someone with a decent knowledge of computing technology to understand and implement these systems. Yet a very specialized level of expertise is requisite for developing these systems and reasoning why they meet a specific security goal. In the last few decades, the field of cryptography has come a long way in understanding the principals and laying down the framework of specifying security goals and proof methods on a firm formal footing. As a consequence of a vast body of work in this field, there is a compact set of techniques that have emerged as the fundamental building blocks of these systems and the reasoning principles behind their security.

Concomitant to the maturity of this field, a substantial research community has grown around attempting to consolidate and automate these reasoning principles so that that the manual need to provide tedious and cumbersome proofs is removed. Importantly, automation provides greater assurance since all possible corner cases are also considered eliminating sometimes subtle errors. Efforts in this direction has led to some very exciting research bringing the attention of experts in such diverse fields of computer science as logic and programming languages to cryptographic systems.

In this work, we took a purely algebraic approach to the problem. Although, there have been many pieces of work in formal methods for cryptographic protocols [AR00, CH, MW04, DDMR07b], this to our knowledege is a novel approach to theorem proving of security protocols. The central feature of our approach is that we consider operations at the highest level of granularity compared to all earlier approaches. Most earlier approaches treat encryptions, signatures etc as basic primitives - whereas in our approach we can look at these operations at various level of abstractions.

So far, the bulk of our work has been to understand the fundamental limits of the concept of simulatability as applied to algebraic systems. At the outset of this project we formally defined the notion of simulatability for very simple pieces of code without any loop or arithmetic operation. The surprising observation, as we saw in Section 10.6, was that even with these simple straight line programs, simulatability is undecidable if unbounded storage is allowed.

The next question we asked was suppose we make the functionalities stateless, but support arithmetic operations, can we have decision procedures for simulatability in these scenarios? Motivated by cryptographic protocols such as commitment schemes, it seemed that an attractive choice would be to consider bitstring objects with support for bitwise xor. Technically, the objects would be from finite fields of characteric two with support for the addition operation. Without any conditional operator, the problem is easily seen to be straightforward to solve. At this point these primitives can model some of the simplest cryptographic protocols such as the one-time pad encryption.

Surprisingly, introduction of conditional operators led to a whole new level of complexity. In Section 10.4, we defined *Pseudo-Linear* functions, which are functions computed by branching programs over the field elements. The conditionals in such programs are built

from equality constraints over linear expressions, closed under negation and conjunction. We found that these functions are *complete* in a very elegant sense: if a given pseudo-linear function is an arbitrary function of a given set of pseudolinear functions, then it is in fact a pseudo-linear function of the given set of functions! The implication of this theorem for a UC system described in terms of pseudo-linear functions is very direct: if there is a simulator which proves UC security of the system, then there is a simulator independent of the security parameter; otherwise the system is not UC secure. In addition, the structure of these special simulators make them straightforwardly enumerable, thus reducing the problem of finding a security proof to a search problem.

Some of the simplest cryptographic security notions require the generation of random numbers. Unguessability of bitstrings generated randomly from large distributions is a basic building block of almost all non-trivial cryptographic primitives. In essence, unguessability is the most primitive and simplest security property from which all higher level properties are boot-strapped. We added random number generation to our set of operations and observed a crucial property in the context of pseudo-linear functions: any randomized pseudo-linear function is statistically indistinguishable from a randomized pseudo-linear function generating just *one* random number - a class of functions we call simple randomized pseudo-linear (SRPL) functions. Our techniques for pseudo-linear functions extended naturally to cover the richer set of SRPL functions and hence we had a decision procedure for the richer language.

So far we had not modeled the interactivity of the ideal functionalities in the sense of a distributed computation. This was the next hurdle we attempted to cross. In this setting it is possible for the simulator to supply additional parameters to the ideal functionilities of its own choosing. The functions computed depend on the parameters supplied by the environment as well as the simulator. There is a fair bit of additional complexity that arises in this setting. The simulator can iteratively compute *any* function of the outputs it has observed so far and supply back the result to any functionality of its choice. To model this, we defined *iterated* SRPL functions. The core insight in the space of these functions was that the action of any probabilistic poly-time algorithm computing these iterates is equivalent to a specific class of iterates which are *finite* in number. Specifically, these special iterates can be automatically enumerated and the order of their set is again independent of the security parameter! This independence is peculiar in this setting since we demonstrated that it is essential to restrict the simulator to be probalistic poly-time bounded in the security parameter, otherwise security may not hold. However, if there does exist such a simulator, then there exists a simulator which is *independent* of the security parameter.

Proceeding even further, we added bounded persistent states across call of functionalities. This is an essential step since the definition of ideal functionalities of many cryptographic primitives require to carry state from invocation to invocation. This state may be in the form of boolean values representing phase transitions internal to a protocol or records of values generated or patterns that may be required to match. A case in point is the ideal functionality for signatures which records each signed message in a lookup table. A valid signature is one which has an entry in the table. As we see in Section 10.6, table lookups,

if unrestricted, can make the problem undecidable. However, cryptographic protocols use signatures and public-key encryption primitives in standard ways. This encouraged us to look for restrictions that enable decision procedures, while covering standard usage. In view of this we first constructed stateful iterated compositions of randomized pseudo-linear functionalities. The key observation in this space was that there is only a finite number of equivalence classes of these iterates modulo permutation of uniformly random quantities and hence represent the *same randomized algorithm*. Again this observation led to an automatic enumeration of possible simulators and thus a decision procedure. At this point many complex primitives can be expressed in the operators allowed, such as password-based key exchange protocols, commitment schemes and so on.

Cryptography aside, there has been a few emerging efforts to model systems security in the UC framework, such as our work on file systems and hypervisors. In contrast to cryptographic protocols, which usually have a fixed number of interactions per session, systems can have unbounded number of interactions. Our work so far allows an arbitrary number of interactions in the ideal side, but the real side is restricted to be monolithic. Note that we can still hope to prove security in the bounded and non-adaptive setting by simple enumeration of possible interaction sequences. Lifting the restriction seems to be a difficult problem in the space of the rich set of operators that we have outlined. However, systems security primitives are built mostly out of access control. So decision procedures may be possible in a weakened language setting.

We have charted a portion of the boundary between what is provably undecidable and provably decidable. The problem is rich in possibilities of further exploration. The hope is that once the number of operations covered are rich enough, we will be able to express and automatically analyze an unprecedented spectrum of protocols and specifications. Some primitives that seem to be on the horizon are hash proof systems, zero knowledge proof systems, pseudo-random functions and so on. Although we have come a long way in adding more and more operators, we still find that the existence of a simulator implies existence of a simulator independent of the security parameter. This is a tantalizing prospect offered by the gap between the decidable and the undecidable: where the shift occurs seems like a fundamental question.

11 Conclusions

We believe that the results obtained in the Montage project are convincing evidence that it is feasible to design software systems in a principled manner with the UC Framework, which can guarantee that the resulting systems are secure and will remain so when composed with other systems. We have successfully shown examples from very diverse areas where the UC framework can be applied outside its traditional domain of cryptographic protocols. The results obtained in the Montage project have had impact on many open source packages. Further, a number of papers and publications have resulted from the Montage effort

11.1 Key Areas of Application

As we have documented in this report, we have been successful in applying the UC methodology to a diverse set of applications. Specifically, we have had success in the following areas

- **Safe subsets of the POSIX filesystem interface** In Sections 5 and 6 we described the application of the UC Framework to the modeling of safe subsets of the POSIX filesystem. We were successful in defining a safe subset which is designed to prevent a large class of attacks. We described an implementation of a safe filesystem primitive and its evaluation on practical systems. Our evaluation shows that our primitive can be a drop-in replacement for the usual primitives and the evaluation also showed that several open source packages have latent vulnerabilities all of which can be prevented by using our primitive.

- **Secure Virtualization primitives** The second application of the UC Framework is to the design of secure virtualization primitive is described in Section 7. We used the UC Framework to model the notion of strong isolation of tenant workloads and we identified specific conditions which are sufficient to guarantee strong isolation. We have also investigated how practical hypervisors can achieve these sufficient conditions.

- **Web Security Protocols** Another area where we have applied the UC Framework is to formally model web security protocols. Specifically, we have investigated the security of the OAuth web security protocol. We have shown a formal proof of correctness of this protocol and also identified a number of practical recommendations on how this protocol should be implemented. This work is described in Section 9.

- **Proof Automation** We initiated a very ambitious effort to understand how the proofs of equivalence required to apply the UC Framework can be automated. We demonstrated a number of positive as well as negative results in this effort. While it is easy to see that the general problem is undecidable, we have shown surprisingly that several very restricted models are also undecidable. On the positive side, we have demonstrated several restricted cases, including several which can model real crypto-

graphic primitives, where the proofs can be automated. These results are described in Section 10.

11.2 Papers and Publications

The following are some of the papers and publications that arise from the Montage project

- Suresh Chari, Shai Halevi, Wietse Venema. *Where Do You Want to Go Today? Escalating Privileges by Pathname Manipulation*. Proceedings of the Network and Distributed systems security 2010.
- Ran Canetti, Suresh Chari, Shai Halevi, Birgit Pfitzmann, Arnab Roy, Michael Steiner, Wietse Venema. *Composable Security Analysis of OS Services* Proceedings of the Applied Cryptography and Network Security 2011.
- Charanjit Jutla and Arnab Roy. *Relatively-Sound NIZKs and Password-Based Key-Exchange*. Proceedings of Public Key Cryptography (PKC) 2012.
- Suresh Chari, Charanjit S. Jutla, Arnab Roy: *Universally Composable Security Analysis of OAuth v2.0* IACR Cryptology ePrint Archive 2011: 526 (2011).
- Charanjit S. Jutla, Arnab Roy: *A Completeness Theorem for Pseudo-Linear Functions with Applications to UC Security* Electronic Colloquium on Computational Complexity (ECCC) 17: 92 (2010)
- Suresh Chari; Charanjit Jutla. *Universally Composable Web Security Protocols for Delegation*. IBM Technical Research Report RC24856.
- Charanjit Jutla, Arnab Roy: *Decision Procedures for Simulatability*. To appear in Proceedings of ESORICS 2012.
- Arnab Roy, Arvind Seshadri, Suresh Chari, Mihai Christodorescu, Dimitrios Pendarakis and Wietse Venema. *noLeak Provable Isolation for Commodity Virtualization Platforms* In Preparation.
- Arvind Seshadri, Arnab Roy, Ning Qu, Adrian Perrig. *Outpost: Creating Secure Execution Environments Without Secure Hardware*. In Preparation.

11.3 Contributions to Open Source

Our evaluation of the safe-open filesystem primitive on multiple UNIX systems revealed a number of vulnerabilities in multiple open source software packages. We have transferred our findings to the following software packages

- A latent privilege escalation vulnerability was reported for the Common UNIX Printing System (CUPS) as bug number 3510 (`http://www.cups.org/str.php?L3510`).

  Here, the privileged CUPS server saves state and overwrites files as `root` in a directory `/var/cache/cups` that is writable by unprivileged helper processes. If an unprivileged CUPS helper program has a vulnerability, then an attacker could escalate privilege by replacing a CUPS state file by a symlink to a sensitive file, and causing symlink target to be overwritten by a `root`-privileged process.

  The CUPS maintainers created a fix that uses a variant of our "safe open" function to protect CUPS programs against symlink or hardlink attacks, and that will be merged into the next release. This technology transfer is completed.

- A latent privilege escalation vulnerability was reported for Fedora Core 12 and earlier releases as bug number 581884 (`https://bugzilla.redhat.com/show_bug.cgi?id=581884`).

  Here, the `/var/lock` directory is writable by unprivileged processes with the "lock" group ID, for example, processes that execute the set-gid `/usr/sbin/lockdev` command. If this program has a vulnerability, then an attacker could replace the directory `/var/lock/subsys` by a symlink to a directory with critical files, such as `/etc`. With this, the `/etc/init.d/killall` script would remove many files under `/etc` as root, when the system shuts down or reboots.

  Fixing this requires changes to the way that scripts in `/etc/init.d` maintain their state files.

Also our investigation of the STARTTLS protocol composition bug resulted in the discovery of the same vulnerability in multiple software packages.

12 References

[* *08] * * *. Linux-VServer, 2008.

[AARR02] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In Burton S. Kaliski Jr., Cetin K. Koc, and Christof Paar, editors, *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems (CHES'02)*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45, 2002.

[AcKKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In Masayuki Abe, editor, *Proc. (CT-RSA'07)*, volume 4377 of *Lecture Notes in Computer Science*, pages 225–242, 2007.

[AHFG10] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proc. CCSW*, pages 103–108, New York, NY, USA, 2010. ACM.

[AR00] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (the computational soundness of formal encryption). In *IFIP International Conference on Theoretical Computer Science (IFIP TCS2000)*, Lecture Notes in Computer Science, Sendai, Japan, August 2000. Springer Verlag.

[ARJS07] Nidhi Aggarwal, Parthasarathy Ranganathan, Norman P. Jouppi, and James E. Smith. Isolation in commodity multicore processors. *IEEE Computer*, 40(6):49–59, 2007.

[BBF+11] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.*, 33(2), 2011.

[BD96] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 2(2):131–152, 1996.

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC*, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[Bib77] Kenneth Biba. Integrity considerations for secure computer systems. Report TR-3153, MITRE, 1977.

[Bis95] Matt Bishop. Race conditions, files, and security flaws; or the tortoise and the hare redux. Report CSE-95-8, Univ. of California at David, 1995.

[BJSW05] Nikita Borisov, Robert Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: how to abuse atime. In *Proc. 14th USENIX Security Symposium*, pages 303–314, 2005.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: A password-based protocol secure against dictionary attacks and password file compromise. In *ACM Conference on Computer and Communications Security*, pages 244–250, 1993.

[BP02] Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. In *Proc. ESORICS*, pages 1–23, 2002.

[BP03] Michael Backes and Birgit Pfitzmann. Intransitive non-interference for cryptographic purposes. In *Proc. Oakland Security and Privacy*, pages 140–, 2003.

[BP04] Michael Backes and Birgit Pfitzmann. Computational probabilistic noninterference. 3(1):42–60, 2004.

[BPR00] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In *EUROCRYPT*, pages 139–155, 2000.

[BPW07] Michael Backes, Birgit Pfitzmann, and Michael Waidner. The reactive simulatability (rsim) framework for asynchronous systems. *Inf. Comput.*, 205(12):1685–1720, 2007.

[Can] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. A revised full version (2005) is available at IACR Eprint Archive, `http://eprint.iacr.org/2000/067/` and at the ECCC archive, `http://eccc.uni-trier.de/eccc-reports/2001/TR01-016/`.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[Can06] Ran Canetti. Security and composition of cryptographic protocols: A tutorial. *SIGACT News*, 37(3 & 4), 2006.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO*, pages 19–40, 2001.

[CG10] Ran Canetti and Sebastian Gajek. Universally composable symbolic analysis of diffie-hellman based key exchange. Cryptology ePrint Archive, Report 2010/303, 2010. `http://eprint.iacr.org/`.

[CGJ09] Xiang Cai, Yuwei Gui, and Rob Johnson. Exploiting unix file-system races via algorithmic complexity attacks. In *IEEE Symposium on Security and Privacy*, pages 27–41, 2009.

[CH] R. Canetti and J. Herzog. Universally composable symbolic analysis of cryptographic protocols (the case of encryption-based mutual authentication and key-exchange). Extended version at `http://eprint.iacr.org/2004/334`.

[CHK+05] Ran Canetti, Shai Halevi, Jonathan Katz, Yehuda Lindell, and Philip D. MacKenzie. Universally composable password-based key exchange. pages 404–421, 2005.

[Cri03] M. Crispin. Internet Message Access Protocol - Version 4rev1. RFC 3501, Internet Engineering Task Force, March 2003.

[CWD02] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.

[cyr11] Bug 3424 - STARTTLS plaintext command injection vulnerability (Cyrus). `http://bugzilla.cyrusimap.org/show_bug.cgi?id=3424`, March 2011.

[DA99] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2246, Internet Engineering Task Force, January 1999.

[DDMR07a] A. Datta, A. Derek, J. C. Mitchell, and A. Roy. Protocol composition logic (PCL). In *Electronic Notes in Theoretical Computer Science*, 2007.

[DDMR07b] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol composition logic (pcl). *Electr. Notes Theor. Comput. Sci.*, 172:311–358, 2007.

[Den76] Dorothy E. Denning. A lattice model of secure information flow. *Comm. ACM*, 19(5):236–243, 1976.

[Den11] Frank Denis. Pure-FTPd 1.0.30 has been released. `http://www.pureftpd.org/project/pure-ftpd/news`, March 2011.

[DFGK09] Anupam Datta, Jason Franklin, Deepak Garg, and Dilsun Kirli Kaynar. A logic of secure systems and its application to trusted computing. pages 221–236, 2009.

[DH04] D. Dean and A. J. Hu. Fixing races for fun and profit: how to use access(2). In *Proc. 13th USENIX Security Symposium*, pages 195–206, 2004.

[Dij75] Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, Winter 1975.

[Dro97] R. Droms. Dynamic host configuration protocol. RFC 1541, Internet Engineering Task Force, March 1997.

[Duf09] Loic Duflot. Getting into the SMRAM: SMM Reloaded. `http://cansecwest.com/csw09/csw09-duflot.pdf`, 2009.

[Fea06] C. Feather. Network News Transfer Protocol (NNTP). RFC 3977, Internet Engineering Task Force, October 2006.

[Fer07] Peter Ferrie. Attacks on virtual machine emulators, 2007.

[FH05] P. Ford-Hutchinson. Securing FTP with TLS. RFC 4217, Internet Engineering Task Force, October 2005.

[FWF09] Leo Freitas, Jim Woodcock, and Zheng Fu. Posix file store in z/eves: An experiment in the verified software repository. *Science of Computer Programming*, 74(4):238–257, February 2009.

[GA03] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *Proc. Oakland Security and Privacy*, pages 154–165, 2003.

[Gar03] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *NDSS*, 2003.

[GAWF07] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proc. 11th Workshop on Hot Topics in Operating Systems (HotOS-XI)*, May 2007.

[GM82] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proc. Oakland Security and Privacy*, pages 11–20, 1982.

[GPS05] Thomas Gro(s), Birgit Pfitzmann, and Ahmad-Reza Sadeghi. Browser model for security analysis of browser-based protocols. In *ESORICS*, pages 489–508, 2005.

[Gre] Greenhills Software, Inc. Integrity real-time operating system.

[Gut96] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proc. USENIX Security*, pages 77–89, 1996.

[HL10] E. Hammer-Lahav, Ed. The OAuth 1.0 protocol. http://tools.ietf.org/html/rfc5849, 2010.

[Hof02] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. RFC 3207, Internet Engineering Task Force, February 2002.

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proc. USENIX Security*, pages 45–60, 2008.

[IEE08] IEEE Std. 1003.1. The open group base spec. http://www.opengroup.org/onlinepubs/9699919799/, 2008.

[Ima11] Release notes for imail server v11.5. http://docs.ipswitch.com/_Messaging/IMailServer/v11.5/ReleaseNotes/index.htm, 2011.

[JH07] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: build a verifiable filesystem. *Formal Asp. Comput.*, 19(2):269–272, 2007.

[JPRZ04] C. Jutla, A. Patthak, A. Rudra, and D. Zuckerman. Testing low-degree polynomials over prime fields. In *FOCS*, 2004.

[KASZ08] Jingfei Kong, Onur Aciiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In Trent Jaeger, editor, *Proc. CSAW*, pages 25–34. ACM, October 2008.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an os kernel. In *SOSP*, pages 207–220, 2009.

[Kem02] Richard A. Kemmerer. A practical approach to identifying storage and timing channels: Twenty years later. In *Proc. ACSAC*, pages 109–118, 2002.

[ker11] Kerio Technologies Information for VU#555316 (US-CERT). http://www.kb.cert.org/vuls/id/MAPG-8D9M4P, March 2011.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Proc. CRYPTO*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, August 1999.

[Kle08] J. Klensin. Simple Mail Transfer Protocol. RFC 5321, Internet Engineering Task Force, October 2008.

[KLP68] T. Kasami, S. Lin, and W. W. Peterson. New Generalization of the Reed-Muller Codes Part I: Primitive Codes. *IEEE Transactions on Information Theory*, IT-14(2):189–199, March 1968.

[Koh79] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill Inc, 2 edition, March 1979.

[Köp07] Boris Köpf. *Formal Approaches to Countering Side-Channel Attacks*. PhD thesis, ETH Zürich, 2007.

[KR04] T. Kaufman and D. Ron. Testing polynomials over general fields. In *FOCS*, 2004.

[K.S] K.Seidler. The xampp software. http://www.apachefriends.org/.

[KSRL10] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: virtualized cloud infrastructure without the virtualization. In *Proc. 37th Annual International Symposium on Computer Architecture (ISCA'10)*, pages 350–361, 2010.

[Kuh02] Markus G. Kuhn. Optical time-domain eavesdropping risks of CRT displays. In *Proc. Oakland Security and Privacy*, pages 3–18. IEEE Computer Society, 2002.

[KW91] Paul A. Karger and J. C. Wray. Storage channels in disk arm optimization. In *Proc. Oakland Security and Privacy*, pages 52–61, 1991.

[KW00] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd International SANE Conference*, 2000.

[KZB+91] Paul A. Karger, Mary Ellen Zurko, Douglas W. Bonin, Andrew H. Mason, and Clifford E. Kahn. A Retrospective on the VAX VMM Security Kernel. *IEEE Trans. Softw. Eng.*, 17:1147–1165, November 1991.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Lip75] Steven B. Lipner. A comment on the confinement problem. In *Proc. SOSP*, pages 192–196, November 1975.

[LTO+11] Xun Li, Mohit Tiwari, Jason Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In Mary W. Hall and David A. Padua, editors, *Proc. PLDI*, pages 109–120. ACM, June 2011.

[Mai] [8lgm]-advisory-5.unix.mail.24-jan-1992. http://www.8lgm.org/advisories/[8lgm]-Advisory-5.UNIX.mail.24-Jan-1992.html.

[Man03] Heiko Mantel. *A uniform framework for the formal specification and verification of information flow security.* PhD thesis, Universität des Saarlandes, 2003.

[McC87] Daryl McCullough. Specifications for multi-level security and a hook-up property. In *Proc. Oakland Security and Privacy*, pages 161–166. IEEE Computer Society, 1987.

[McL94] John McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. Oakland Security and Privacy*, pages 79–93, 1994.

[McL96] John McLean. A general theory of composition for a class of "possibilistic" properties. *IEEE Trans. Software Eng.*, 22(1):53–67, 1996.

[Mil08] Ken Milberg. Workload Partitioning (WPAR) in AIX 6.1, 2008.

[MK97] David Mazières and M. Frans Kaashoek. Secure applications need flexible operating systems. In *Workshop on Hot Topics in Operating Systems*, pages 56–61, 1997.

[Moc87] P. Mockapetris. Domain names - implementation and specification. RFC 1035, Internet Engineering Task Force, November 1987.

[MR96] J. Myers and M. Rose. Post Office Protocol - Version 3. RFC 1939, Internet Engineering Task Force, May 1996.

[MVN06] K. Murchison, J. Vinocur, and C. Newman. Using Transport Layer Security (TLS) with Network News Transfer Protocol (NNTP). RFC 4642, Internet Engineering Task Force, October 2006.

[MW04] D. Micciancio and B. Warinschi. Completeness theorems for the abadi-rogaway logic of encrypted expressions. *Journal of Computer Security*, 12(1):99–129, 2004.

[Nat95] National Computer Security Center. Final evaluation report of Gemini Computers Incorporated: Gemini Trusted Network Processor Release 1.01. NCSC-FER-94/34, 1995.

[net] The netqmail website. `http://www.netqmail.org`.

[Neu] Peter Neumann. Principled assuredly trustworthy composable architectures. `http://www.csl.sri.com/users/neumann/chats.html`.

[New99] C. Newman. Using TLS with IMAP, POP3 and ACAP. RFC 2595, Internet Engineering Task Force, June 1999.

[NF03] Peter G. Neumann and Richard J. Feiertag. Psos revisited. In *ACSAC*, pages 208–216, 2003.

[nvd] National vulnerability database. `http://nvd.nist.gov/`.

[ope02] Common Vulnerabilities and exposures CVE-2001-0529. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0529`, 2002.

[ope08] The open group base specifications issue 7; ieee std 1003.1-2008. `http://www.opengroup.org/`, 2008.

[Ora11a] Oracle critical patch update advisory - april 2011. `http://www.oracle.com/technetwork/topics/security/cpuapr2011-301950.html`, April 2011.

[Ora11b] Oracle Corporation. Java, 2011.

[Orm07] Tavis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. In *CanSecWest Applied Security Conference*, Vancouver, BC, Canada, April 2007.

[PCI06] PCI SIG. *PCI Express Access Control Services (ACS)*, Oct 2006.

[Per05] Colin Percival. Cache missing for fun and profit. In *BSDCan*, Ottawa, ON, Canada, 2005.

[Plu82] David C. Plummer. An ethernet address resolution protocol. RFC 826, Internet Engineering Task Force, November 1982.

[PR85] J. Postel and J. Reynolds. File transfer protocol (FTP). RFC 959, Internet Engineering Task Force, October 1985.

[PT04] Daniel Price and Andrew Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *Proc. LISA*, pages 241–254. USENIX, November 2004.

[PW00] Birgit Pfitzmann and Michael Waidner. Composition and integrity preservation of secure reactive systems. In Sushil Jajodia, editor, *Proceedings of the 7th ACM Conference on Computer and Communications Security*, pages 245–254, Athens, Greece, November 2000. ACM Press.

[qma11] qmail-tls patch for VU#555316. `http://inoa.net/qmail-tls/vu555316.patch`, March 2011.

[qps11] [smtpd/qpsmtpd] 520024: Fix STARTTLS vulnerability for async. `http://www.nntp.perl.org/group/perl.qpsmtpd.dev/2011/06/msg391.html`, 2011.

[rcs08] `initscripts` arbitrary file deletion vulnerability. cve-2008-3524. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-3524`, 2008.

[RNSE09] Himanshu Raj, Ripal Nathuji, Abhishek Singh, and Paul England. Resource management for isolation enhanced cloud services. In *Proc. CCSW*, pages 77–84. ACM, 2009.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. ACM CCS*, pages 199–212. ACM, 2009.

[Rus82] John M. Rushby. Proof of separability: A verification technique for a class of a security kernels. In *Proc. 5th Colloquium on International Symposium on Programming*, pages 352–367. Springer-Verlag, 1982.

[Sea] M. Seaborn. Plash: tools for practical least privilege. http://plash.beasts.org/.

[SK10] Udo Steinberg and Bernhard Kauer. Nova: a microhypervisor-based secure virtualization architecture. In *Proc. Eurosys*, EuroSys '10, pages 209–222. ACM, April 2010.

[SM07] R. Siemborski and A. Melnikov. SMTP Service Extension for Authentication. RFC 4954, Internet Engineering Task Force, July 2007.

[sol08] Security Vulnerability in inetd(1M) Daemon When Debug Logging is Enabled. CVE-2008-1684. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-1684, 2008.

[spa12] Spamdyke version 4.2.1 changelog. http://www.spamdyke.org/documentation/Changelog.txt, January 2012.

[ST04] Adi Shamir and Eran Tromer. Acoustic cryptanalysis: on nosy people and noisy machines. In *Rump session of Euurocrypt04*. 2004.

[THWS08a] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably preventing file race attacks with user-mode path resolution. Report RC24572, IBM Research, 2008.

[THWS08b] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file tocttou races with hardness amplification. In *Proceedings of* USENIX Conference on File and Storage Technologies (FAST), pages 189–206, 2008.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23:37–71, January 2010.

[TSW08] D. Tsafrir, D. Da Silva, and D. Wagner. The murky issue of changing process identity: revising"setuid demystified". *USENIX:login*, 33(3):55–66, 2008.

[TWM+09] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proc. ASPLOS*, pages 109–120. ACM, 2009.

[TWV+08] Hendrik Tews, Tjark Weber, Marcus Völp, Erik Poll, Marko van Eekelen, and Peter van Rossum. Nova micro–hypervisor verification. Technical Report ICIS–R08012, Radboud Universteit Nijmegen, May 2008.

[Ven] Wietse Venema. The postfix mail transfer agent. http://www.postfix.org.

[Ven11] Wietse Venema. Plaintext command injection in multiple implementations of STARTTLS (CVE-2011-04. http://www.postfix.org/CVE-2011-0411.html, March 2011.

[wat11] Release Notes for WatchGuard XCS v9.1 TLS Hotfix. http://www.watchguard.com/support/release-notes/xcs/9/en-US/EN_ReleaseNotes_XCS_9_1_1/EN_ReleaseNotes_WG_XCS_9_1_TLS_Hotfix.pdf, April 2011.

[WL06] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proc. ACSAC*, pages 473–482. IEEE Computer Society, December 2006.

[WR09] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel(R) CPU Cache Poisoning. http://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf, 2009.

[Wra91] J. C. Wray. An analysis of covert timing channels. In *Proc. Oakland Security and Privacy*, pages 2–7. IEEE Computer Society, 1991.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. *SIGOPS Operating Systems Review*, 36:195–209, December 2002. special issue for Proc. OSDI.

[xte93] CERT Coordination Center. CERT Advisory CA-1993-17. http://www.cert.org/advisories/CA-1993-17.html, 1993.

[ZBA10] Sebastian Zander, Philip Branch, and Grenville Armitage. Estimating the capacity of temperature-based covert channels. Technical Report 100726A, Centre for Advanced Internet Architectures, Swinburne University of Technology, July 2010.

[ZJOR11] Yinqian Zhang, Ari Juels, Alina Oprea, and Michael K. Reiter. HomeAlone: Co-residency detection in the cloud via side-channel analysis. In *Proc. Oakland Security and Privacy*, pages 313–328. IEEE Computer Society, 2011.

List of Symbols, Abbreviations and Acronyms

API Application programming interface

ARP Address Resolution Protocol

CA Certificate Authority

CPU Central Processing Unit

CUPS Common Unix Print Services

DFSRA Deterministic finite-state resource arbiter

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

FIFO First-in First-out policy

FTP File Transfer Protocol

IETF Internet Engineering Task Force

IMAP Internet Message Access Protocol

IOMMU Input/output memory management unit

ISA Instruction Set Architecture

ITM Probabilistic Interactive Turing Machine

LD_PRELOAD Environment variable for dynamic linking

MTA Mail Transfer Agent

NIZK Non-Interactive Zero Knowledge

NNTP Network News Transfer Protocol

OS Operating System

PAK Password Authenticated Key Exchange

PE Processing Element

PKE Public Key Encryption

PKI Public Key Infrastructure

POP Post Office Protocol

POSIX Portable Operating System Interface

PPT Probabilistic Polynomial time

PRA Probabilistic Resource Arbiter

SASL Simple Authentication and Security Layer

SMM System Management Mode

SMTP Simple Mail Transfer Protocol

SRPL Simplified Randomized Pseudo-Linear

SSL Secure Sockets Layer

TDM Time-division multiplexing

TLS Transport Layer Security Protocol

TOCTTOU Time-of-check to time-of-use

TVD Trusted Virtual Domain

UC Universal Composability

$\text{UCC}_{OneTime}$ Commitment Scheme

URI Uniform Resource Identifier

VM Virtual Machine

XAMPP Cross-platform, Apache, MySQL, Perl and PHP

A Adversary in the real-world

$\mathcal{B}$ Set of Roles

C Interface call

EPSELIN Elementary pseudo-linear polynomials

$\mathcal{F}$ Ideal functionality

`fName` Absolute pathname of a file

F_{com} Ideal Commitment Functionality

f_i Pseudo-linear function

F_{pke} Ideal functionality for Public Key Encryption

$\mathcal{F}_q$ Finite field

F_{sig} Ideal functionality for Signature

G_{pk} pseudorandom generator derived from pk

I Input alphabet

$KGen$ Public Key Generation Algorithm

L Language describing an system in ideal or real world

$L^{\$,\oplus,\mathsf{if}}$ Restricted Language to describe Ideal functionalities

$L^{\$,\oplus,\mathsf{if},\mathsf{state}}$ Restricted Language to describe Ideal functionalities with state

L_{tab} Language for expressing functionalities where equivalnece is undecidable

$\mathcal{M}(P)$ Maninpulators of pathname P

msg Message exchanged between parties

O Output alphabet

P Pathname of file

P_i Party in a distributed protocol or system

P_{iso} Isolation Property

pk Public Key

$p(\vec{x})$ Multivariate polynomial

pw Password supplied in a protocol

$\mathcal{Q}(X)$ Set of basic pseudo-linear polynomials in variables X

REPSELIN Reduced elementary pseudo-linear polynomials

$\mathcal{S}$ Simulator in the ideal-world

sid Session ID

SIMPFS Idealized Filesystem Interface

simpfs POSIX implemetation of Idealized filesystem

t Time instant

U User id in a system

$\mathcal{Z}$ Environment the system runs in

$\mathcal{B}$ Basis for a set of polynomials

$\perp$ Indeterminate value

$\exists$ There exists

$\mathcal{F}_{\text{OAUTH}^*}$ OAuth 2.0 Authorization Code Ideal Functionality

$\forall$ For all

$\mathcal{F}_{\text{PWKE}}$ Password-Based Key Exchange Functionality

$\mathcal{F}_{\text{SC}}$ Secure Channel Ideal Functionality

$\mathcal{F}_{\text{SSL}}$ SSL Ideal Functionality

$\mathcal{I}$ Input Trace

π Implementation of an ideal functionality

$\mathcal{P}$ Implementation of an ideal functionality

π_{VM_i} Projection function of machine state to VM_i

s_0 Start State

Σ Finite set of states

$\sigma(t)$ Machine state at time t